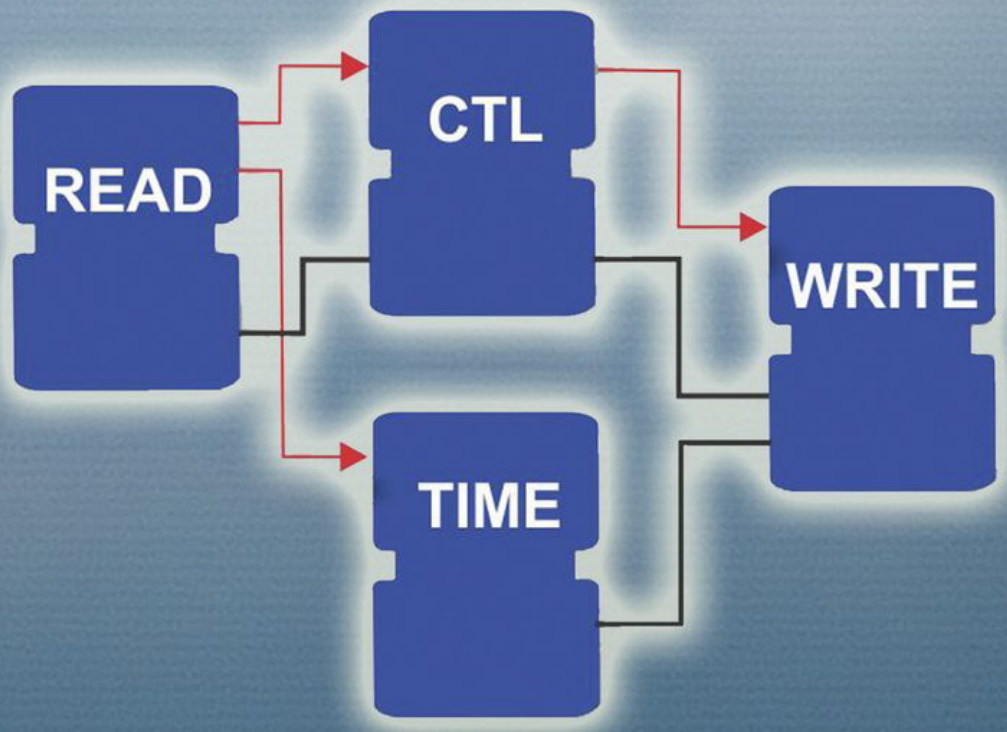
В. Н. Дубинин, В. В. Вяткин

МОДЕЛИ ФУНКЦИОНАЛЬНЫХ БЛОКОВ IEC 61499, ИХ ПРОВЕРКА И ТРАНСФОРМАЦИИ В ПРОЕКТИРОВАНИИ РАСПРЕДЕЛЕННЫХ СИСТЕМ УПРАВЛЕНИЯ

Монография



МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
«Пензенский государственный университет» (ПГУ)

В. Н. Дубинин, В. В. Вяткин

Модели функциональных блоков IEC 61499, их проверка и трансформации в проектировании распределенных систем управления

Монография

Под редакцией профессора Н. П. Вашкевича

Пенза
Издательство ПГУ
2012

УДК 681.5; 004.4
Д79

Р е ц е н з е н т ы:
лаборатория автоматизации отдела электроники
и автоматизации ГНЦ Института физики высоких энергий
(начальник лаборатории – доктор физ.-мат. наук, проф. В. Н. Алферов);
доктор технических наук, профессор, заведующий кафедрой электротехники
и мехатроники Технологического института
Южного федерального университета в г. Таганроге
В. Х. Пшихопов

Дубинин, В. Н.

Д79 Модели функциональных блоков IEC 61499, их проверка и трансформации в проектировании распределенных систем управления : моногр. / В. Н. Дубинин, В. В. Вяткин ; под ред. проф. Н. П. Вашкевича. – Пенза : Изд-во ПГУ, 2012. – 348 с.

ISBN 978-5-94170-521-4

Освещаются вопросы разработки элементов теории и технологии проектирования распределенных компонентно-базированных систем управления промышленной автоматики нового поколения на основе международного стандарта IEC 61499. Рассматриваются операционная семантика функциональных блоков IEC 61499 для различных моделей выполнения, метод формальной верификации и метод семантического анализа проектов IEC 61499 на основе Web-онтологий. Предлагается унифицированный подход к проектированию систем управления на основе трансформации графов. В рамках данного подхода решается задача синтеза формальных моделей, рефакторинга и портабельности управляющего программного обеспечения.

Предназначена для специалистов в области вычислительной техники, автоматики, робототехники.

УДК 681.5; 004.4

ISBN 978-5-94170-521-4

СОДЕРЖАНИЕ

Введение ... 6

1. Обзор и анализ методов проектирования современных распределенных систем управления промышленными процессами ... 12

1.1. Требования, тенденции развития и особенности современных систем промышленной автоматики ... 12

1.2. Обзор стандарта IEC 61499 ... 16

1.3. Модели выполнения ... 22

1.4. Формализованное описание и верификация систем управления на основе IEC 61499 ... 36

1.5. Методы проектирования систем управления на основе IEC 61499 ... 53

2. UML-FB – визуальный язык для моделирования систем управления промышленными процессами на основе стандарта IEC 61499 ... 59

2.1. Краткое описание языка UML-FB ... 59

2.2. Разработка UML-FB-спецификации производственной системы FESTO ... 69

2.3. Трансформация UML-моделей в функциональные блоки ... 76

2.4. Разработка визуальных имитационных моделей мехатронных компонентов ... 78

2.5. Графовые шаблонные запросы ... 84

3. Операционная семантика ФБ ... 87

3.1. Системная конфигурация ... 87

3.2. Развертывание системной конфигурации ... 89

3.3. Буферирование данных ... 92

3.4. Переход от иерархических структур систем ФБ к одноуровневым ... 94

3.5. Модульная формальная модель операционной семантики ФБ ... 96

3.6. Функционально-структурная организация моделей систем ФБ ... 100

3.7. Базовая модель базисного функционального блока ... 104

3.8. Модель составного функционального блока для циклической модели выполнения ... 118

3.9. Модель диспетчера для циклической модели выполнения ... 121

3.10. Взаимосвязь модулей функциональных блоков ... 123

3.11. Модель составного функционального блока для синхронной модели выполнения ... 125

3.12. Модель диспетчера для синхронной модели выполнения ... 128

3.13. Семантика последовательной модели выполнения ... 133

3.14. Формальная модель системы ФБ в виде системы переходов состояний ... 144

4. Проверка моделей систем функциональных блоков.....155
4.1. Подход на основе отображения машин абстрактных состояний в модули *SMV* (подход 1) 157
4.2. Подход на основе использования структуры Крипке и предложений *TRANS* в *SMV* (подход 2) 161
4.3. Пример. Модель системы двух АЛУ 166
4.4. Демонстрация подхода 1 167
4.5. Демонстрация подхода 2 176
5. Графотрансформационный подход к синтезу формальных моделей систем функциональных блоков.....181
5.1. Краткие сведения из области трансформации графов 181
5.2. Поток моделей в процессе синтеза.....182
5.3. Моделирование систем ФБ на основе арифметических *NCES*-сетей 184
5.4. Метамодели исходной и целевой модельных форм.....195
5.5. Правила перехода от многоуровневой структуры систем функциональных блоков к одноуровневой структуре.....202
5.6. Правила синтеза многоуровневых *aNCES*-сетей на основе одноуровневых систем функциональных блоков 206
5.7. Пример. Синтез сетевой модели для функционального блока «RS-триггер» 213
5.8. Реализация системы синтеза формальных моделей функциональных блоков.....216
6. Рефакторинг диаграмм управления выполнением базисных функциональных блоков 217
6.1. Модель диаграммы управления выполнением *ECC* 217
6.2. Модели выполнения *ECC*.....218
6.3. Общий подход к рефакторингу и исправлению диаграмм *ECC* 219
6.4. Рефакторинг на основе графотрансформационного подхода 224
6.5. Примеры рефакторинга *ECC* 228
6.6. Реализация системы рефакторинга в *AGG* 232
6.7. Оценка системы рефакторинга.....235
7. Семантический анализ проектов IEC 61499 на основе Web-онтологий 244
7.1. Онтология функциональных блоков.....245
7.2. Свойства, связанные с семантической корректностью описаний 263
7.3. Обнаружение циклов с использованием графов зависимостей 273
7.4. Графы зависимостей для алгоритмов 283
7.5. Вопросы сложности логического вывода.....284

7.6. Система семантического анализа проектов IEC 61499 284
8. Шаблоны модельно-ориентированной реализации систем функциональных блоков стандарта IEC 61499 286
8.1. Общее описание метода 287
8.2. Трансформация базисных функциональных блоков 288
8.3. Буферирование сигналов 295
8.4. Трансформация составных функциональных блоков 296
8.5. Трансформация СИФБ 303
8.6. Шаблон «Циклическая модель выполнения» 304
8.7. Шаблон реализации «Синхронная модель выполнения» 309
8.8. Примеры применения шаблонов 312
8.9. Оценка сложности 317
Заключение 320
Библиографический список 321
Summary 345

Введение

Современные системы управления в сфере промышленной автоматики (как правило, построенные на основе международного стандарта IEC 61131-3) являются централизованными системами со всеми вытекающими отсюда негативными последствиями. К ним можно отнести низкую надежность и производительность, сложность настройки и поддержки, сложность модификаций (наращивания, удаления и изменения компонентов) и построения реконфигурируемых систем, проблемы с масштабируемостью, повторным использованием компонентов, а также дороговизной как самого процесса проектирования, так и всей (реальной) системы. В то же время все более жесткие условия, в которые рынок промышленных товаров ставит производителей (в числе которых переход от массового производства к мелкосерийному и штучному производству, необходимость быстрой перенастройки оборудования на выпуск новой продукции, усложнение выпускаемых изделий, повышение требований к ним и т.д.), подталкивают их к переходу на новый технологический базис, позволяющий существенно сократить время проектирования и перепроектирования систем управления и иметь возможность их быстрой реконфигурации.

Принятый в 2005 г. новый международный стандарт IEC 61499, нацеленный на построение распределенных систем управления промышленными процессами, призван решить вызовы времени. По сути дела, стандарт IEC 61499 вводит класс систем управления *нового поколения*. Это интеллектуальные реконфигурируемые распределенные компонентно-базированные системы. Стандарт IEC 61499 поддерживает парадигму проектирования на основе функциональных блоков (ФБ). Эта парадигма вобрала в себя черты компонентного, объектно ориентированного и автоматного подходов к проектированию и программированию сложных управляющих систем. В определенной мере системы управления на основе ФБ можно отнести к системам логического управления, однако область их применения не ограничивается только рамками логического управления вследствие наличия в ФБ мощной обрабатывающей компоненты в виде алгоритмов, оперирующих данными самых разнообразных типов. ФБ являются основными артефактами проектирования в стан-

дарте IEC 61499. Особенностями ФБ являются: машина состояний потоков событий (называемая диаграммой *ECC – Execution Control Chart*), входные и выходные событийные переменные (с собственным управлением) для представления потоков управления, входные и выходные переменные для представления потоков данных, а также ориентация функциональных блоков на реализацию (ФБ – выполнимая спецификация). Установка (*sampling*) значений входных и выходных переменных производится с помощью событийных входов и выходов соответственно. Для этого используются специальные *WITH*-связи. С состояниями *ECC* могут быть связаны выходные события и алгоритмы, определенные на одном из языков программируемых логических контроллеров стандарта IEC 61131-3. В соответствии со стандартом IEC 61499 управляющее приложение представляется в виде сети связанных между собой ФБ, которые могут выполняться на различных ресурсах и устройствах системы.

Работы в области ФБ IEC 61499 интенсивно ведутся во всем мире. В этих работах участвуют научные коллективы и исследователи как из академических учреждений, так и из коммерческих фирм. Однако многие проблемы, связанные со стандартом, остались нерешенными. Среди них можно выделить следующие: 1) разработка методов и средств проектирования и повторного проектирования распределенных систем управления на основе IEC 61499; 2) разработка удобных моделей выполнения и обеспечение портабельности управляющего программного обеспечения (ПО) на основе IEC 61499; 3) разработка методов миграции проектов, основанных на «старом» стандарте IEC 61131-3, поддерживающем концепцию ПЛК, на новую платформу IEC 61499.

По первой проблеме не до конца решены вопросы по формализации систем управления на основе IEC 61499, по стандартизации и формализации моделей выполнения ФБ, по методам проектирования систем управления, по их эффективному описанию, рефакторингу, верификации, реализации и т.п. Методы проектирования систем управления на основе ПЛК в своем большинстве оказались непригодными для проектирования систем управления на основе IEC 61499.

Одним из наиболее перспективных направлений в сфере проектирования программного обеспечения является подход на основе управления моделями (*Model-Driven Engineering*). Данный подход еще не нашел должного развития в сфере программирования на основе IEC 61499, но он уже начинает набирать обороты. В качестве

примера можно сослаться на европейский проект *MEDEIA* (*Model-Driven Embedded System Design Environment for the Industrial Automation Sector*,), стартовавший в 2008 г.

Стандарт IEC 61499 не нашел еще должного применения на практике. Это в большой мере определяется тем, что в нем заложен семантический разрыв между моделью ФБ и моделями выполнения ФБ. Часто это определяют как наличие «дыр» в семантике ФБ. Модель ФБ допускает целый ряд моделей выполнения. К настоящему времени накоплен довольно большой объем теоретических и практических знаний в области моделей выполнения ФБ. Каждая из известных моделей выполнения имеет свои преимущества и недостатки и в той или иной мере удовлетворяет многочисленным, порой противоречивым требованиям, предъявляемым к модели выполнения со стороны разработчиков систем управления. Таким образом, системы управления, разработанные на основе ФБ, в настоящее время не обладают свойством портабельности. Многообразие моделей выполнения определяет интерес к разработке таких шаблонов проектирования, которые обеспечивали бы независимость поведения системы ФБ от используемой модели выполнения. Подобные шаблоны в мировой практике не разрабатывались.

Объектом исследования являются распределенные системы управления промышленной автоматики нового поколения, построенные на основе новых международных стандартов, прежде всего IEC 61499. *Целью данной работы* является разработка эффективных методов и средств проектирования надежных систем управления промышленными процессами нового поколения.

Данная работа структурирована следующим образом. В *первом разделе* дается анализ методов проектирования систем управления на основе стандарта IEC 61499, в том числе методов формализованного описания и методологий проектирования. В начале раздела приводится краткий обзор тенденций развития современных систем промышленной автоматики, вызовов в этой сфере, требований, предъявляемых к системам управления. Дается обзор стандарта IEC 61499, определяются его преимущества, особенности и проблемы. Довольно подробно рассматриваются модели выполнения ФБ, в числе которых как «классические», так и «неклассические». К классическим моделям относятся циклическая и синхронная модели выполнения, модель на основе последовательной гипотезы, *NPMTR*-модель. К неклассическим моделям относятся произвольные модели, определяемые параметризацией и с помощью механизма спусковых функций.

Второй раздел посвящен разработке и использованию *UML-FB* – визуального языка для моделирования систем управления промышленными процессами на основе стандарта IEC 61499. Этот язык, основанный на «легком» расширении метамодели языка *UML* с использованием механизма стереотипов, сокращает семантический разрыв между структурным и объектно ориентированным подходами к проектированию систем управления. Приводится пример спецификации системы управления производственной установкой *FESTO* с использованием языка *UML-FB*, подтверждающий пригодность и удобство языка *UML-FB*. Кратко рассматриваются правила прямой трансформации *UML-FB*-описаний в стандартное представление систем ФБ, а также правила обратной трансформации стандартного представления систем ФБ в *UML-FB*-описания. Предложен подход с использованием языка *UML-FB* (в рамках шаблона проектирования *Model/View/Controller*) для проектирования визуальных имитационных моделей мехатронных компонентов. Данный подход проиллюстрирован рядом примеров. В конце второго раздела рассматриваются вопросы создания языка графовых шаблонных запросов на основе *UML-FB* для систем на основе ФБ. Данный язык может использоваться при создании поисковых систем и репозитариев ФБ.

В *третьем разделе* решается проблема определения операционной семантики ФБ. В начале раздела рассматриваются вопросы развертывания системной конфигурации до уровня экземпляров и буферирования данных. В основной части раздела предлагаются две семантики ФБ: первая основана на аппарате машин абстрактных состояний (МАС), а вторая – на системах переходов состояний (СПС). В рамках первого подхода определяется так называемая модульная формальная модель операционной семантики ФБ (ФМОСФБ), представляющая модифицированную МАС, которая используется в дальнейшем в качестве формальной нотации, а также функционально-структурная организация моделей систем ФБ. С использованием первой формальной нотации описывается семантика базисных и составных ФБ для основных моделей выполнения: циклической, синхронной и последовательной. Формальная модель системы ФБ определяет как схему модели, представляющую набор переменных и набор функций для изменения значений переменных, так и динамику модели в виде правил изменения этих функций. Использование СПС проиллюстрировано на примере определения семантики ФБ для циклической модели выполнения.

В *четвертом разделе* адаптируется метод *Model Checking* для формальной верификации систем ФБ. Предложены методики коди-

рования формальных моделей систем ФБ на языке верификатора *SMV*, построенные соответственно на основе двух различных подходов: а) на основе формальной метамодели ФБ, разработанной с использованием ФМОСФБ; б) на основе формальной метамодели ФБ, разработанной с использованием СПС. Предложенные методики проиллюстрированы на конкретном примере системы ФБ, в верификаторе *SMV* произведена проверка исследуемой системы примера, доказан ряд свойств.

Пятый раздел посвящен разработке графотрансформационного подхода к синтезу формальных моделей систем функциональных блоков. В качестве формальных моделей используются арифметические *NCES*-сети (*aNCES*-сети). В начале раздела рассматриваются теоретические основы графовых трансформаций. В основной части предлагается обобщенный граф потока моделей, используемый в процессе синтеза сетевых моделей систем ФБ. Данный граф включает множество моделей и их трансформаций. Отправным пунктом преобразований является графовая модель исходной системы ФБ, а конечным – графовая модель одноуровневой *aNCES*-сети. Описываются метамодели систем ФБ и *aNCES*-сетей разных классов в виде типизированных атрибутных графов. Рассматриваются правила трансформации графов для синтеза формальных моделей управляющих приложений IEC 61499, включающие правила перехода от многоуровневой структуры систем ФБ к одноуровневой структуре и правила синтеза многоуровневых *aNCES*-сетей на основе одноуровневых систем ФБ. Предлагается прототип системы синтеза формальных моделей систем ФБ, ядром которой является система трансформации графов *AGG*. В конце раздела разработанная система правил трансформации оценивается на конкретном примере.

В *шестом разделе* рассматривается рефакторинг диаграмм *ECC* базисных ФБ, предназначенный для избавления их от условных дуг и удаления потенциально тупиковых состояний. Для определения процесса рефакторинга используется упрощенная модель диаграммы *ECC*. Предлагается графотрансформационный подход к рефакторингу диаграмм *ECC*. В рамках данного подхода разработана система рефакторинга, включающая графовую метамодель *ECC* и правила трансформации (перезаписи) графов. Описывается прототип системы рефакторинга на основе инструментального средства трансформации графов *AGG*. Предлагается набор тестовых *ECC* для оценки системы рефакторинга, включающий более 30 *ECC*. Оцениваются как качественные, так и количественные характеристики

системы рефакторинга. Основным методом для оценки системы рефакторинга был выбран подход, основанный на тестировании.

В *седьмом разделе* определяется онтологический подход к семантическому анализу систем управления, основанных на стандарте IEC 61499. Описывается онтология ФБ, построенная с использованием дескриптивной логики и логики хорновских дизъюнктов. В рамках предложенного подхода разработаны: а) конкретные семантические ограничения, позволяющие выявить многие семантические ошибки в описании систем управления; б) метод обнаружения опасных циклов для иерархических структур систем ФБ. Отмечаются особенности реализации онтологии ФБ на основе языков *OWL DL* и *SWRL* в инструментальной системы *Protégé.* Метод обнаружения опасных циклов демонстрируется на конкретном примере. Предлагается структура системы семантического анализа проектов IEC 61499 на основе *Web*-онтологий. В конце раздела обсуждаются вопросы онтологического представления графов зависимостей в системах ФБ и их использования в процессе проектирования, а также вопросы сложности логического вызова на основе онтологических языков.

В *восьмом разделе* излагается подход к решению проблемы портабельности управляющего программного обеспечения, построенного в соответствии с IEC 61499, на основе шаблонов модельно ориентированной реализации систем функциональных блоков (ШМОРСФБ). Рассматривается трансформация базисного ФБ, включая интерфейс и диаграммы *ECC*, выполняемая в рамках ШМОРСФБ. Трансформация составных ФБ во многом определяется моделью выполнения. Отмечаются место и роль буферизации в реализации шаблонов. Описываются два шаблона реализации, ориентированных на циклическую и синхронную модели выполнения. Предлагаются два примера для оценки применимости шаблонов, первый из которых связан с системой круиз-контроля атомобиля, а второй – с вычислением итерационной суммы. В конце раздела представлены оценки сложности накладных расходов при преобразовании составных и базисных ФБ.

1. Обзор и анализ методов проектирования современных распределенных систем управления промышленными процессами

1.1. Требования, тенденции развития и особенности современных систем промышленной автоматики

Системы промышленной автоматики образуют класс технических систем, в котором интегрированы программные, аппаратные и мехатронные компоненты, тесным образом взаимодействующие друг с другом, что увеличивает сложность системы в целом и определяет междисциплинарный характер предмета. Управляющее программное обеспечение (ПО) является основной частью современных систем промышленной автоматики для обеспечения корректных и безопасных операций процессов автоматизации. Основным стандартом для разработки систем управления промышленной автоматики в настоящее время является международный стандарт IEC 61131-3 [162], ориентированный на проектирование централизованных систем управления на основе программируемых логических контроллеров (ПЛК). Одной из основных целей стандарта IEC 61131-3 была унификация концепции программирования управляющих приложений. Международной электротехнической комиссии (МЭК или IEC) удалось свести все многообразие используемых языков к пяти языкам (*LD*, *IL*, *ST*, *SFC*, *FBD*) [162]. В настоящее время каждый производитель управляющих устройств поддерживает (хотя бы частично) данный стандарт. Этот стандарт вводит концепцию функционального блока (ФБ), инкапсулирующего определенную функциональность. ФБ может содержать один алгоритм, а также данные. Язык ФБ в определенной степени обладает свойствами объектно ориентированных языков, что упрощает повторное использование компонентов, однако IEC 61131-3 не поддерживает такие свойства, как наследование и концепция интерфейсов. Стандарт IEC 61131-3 имеет два недостатка в отношении повторности использования компонент: 1) использование глобальных данных;

2) невозможность прямого управления порядком выполнения ФБ. Следует отметить, что стандарт IEC 61131-3 был принят сравни-

тельно давно (1993 г.) и в нем не учтены новые технологии разработки ПО.

Требования современного рынка к быстрой перенастройке производственных систем для выпуска новых видов изделий, увеличение сложности выпускаемых изделий и соответственно систем управления породили новые проблемы при использовании централизованных систем на основе IEC 61131-3. Оказалось, что отсутствие поддержки передовых методов проектирования ПО увеличивает временные и материальные затраты на разработку сложной системы управления и уменьшает ее надежность. Централизованная архитектура не поддерживает масштабируемость и реконфигурируемость системы и приводит к деградации показателей производительности. Следует заметить, что термин «распределенная обработка» в этих условиях относится только к съему данных с пространственно распределенных датчиков через промышленную сеть и обработки ее в ПЛК.

Ранее предпринимались попытки упростить интеграцию ПЛК в системы, взаимодействующие через коммуникационную сеть с использованием интеграционных компонентных архитектур, таких как *Modbus-IDA* и *PROFInet-CBA* [241]. Однако эти решения оказались полумерами для построения действительно распределенных систем автоматики, где «интеллектуальность» с самого начала проектируется как функциональность, децентрализованная и встроенная в программные компоненты, свободно распределяемые по сетевым устройствам. Как правило, распределенные системы характеризуются структурной и поведенческой сложностью. Показательно, что проектирование распределенных систем относится к числу больших вызовов в области вычислений [158].

С учетом существующих тенденций и достижений в сфере новых технологий разработки ПО и телекоммуникаций в 2005 г. Международной электротехнической комиссией был принят новый стандарт IEC 61499 [163]. Этот стандарт определяет путь для построения систем управления промышленными процессами нового поколения. Это интеллектуальные распределенные компонентно-базированные системы. Архитектура IEC 61499 строится на основе определений IEC 61131-3. Основной элемент архитектуры – снова ФБ, хорошо знакомый и широко используемый инженерами, но его концепция расширена в нескольких направлениях для того, чтобы отразить новые достижения в области проектирования современных программных систем. Одно из основных расширений ФБ – событийный интерфейс, который позволяет явно определить последова-

тельности выполнения ФБ. Другое расширение ФБ связано с объектной и компонентной ориентацией. По аналогии с методами, инкапсулированными в объект, ФБ может содержать несколько алгоритмов. Однако эти алгоритмы являются невидимыми извне, и к ним нет прямого доступа. Кроме того, в ФБ нет глобальных данных. Все это повышает повторную используемость ФБ. ФБ IEC 61499 представляет собой независимую программную единицу, которая может быть реализована, протестирована и использована отдельно от других ФБ.

Инструментальные средства реализации систем управления на основе IEC 61499 делятся на две категории: 1) средства поддержки проектирования (*workbench*) и 2) среды выполнения (*run-time environment*). Примерами некоммерческих средств первого типа являются *FBDK* [134], *4DIAC-IDE* [73], *Corfu/Archimedes* [225] и *FBench* [130], а второго – *FBRT* [134], *FORTE* [73], *FUBER* [97], *CyclicRT* [223]. Первым коммерческим продуктом, поддерживающим IEC 61499 и IEC 61131-3, явилась система *ISaGRAF* (фирма *ICSTriplex*) [160]. Вторая коммерческая система *NxtStudio* (фирма *NxtControl*, Австрия) [187] ориентирована исключительно на IEC 61499. Особенностью последней системы является интеграция распределенной системы управления с системой *SCADA*.

Большинство разработок систем на основе IEC 61499 носят исследовательский характер и не вышли из стен лабораторий. Следует, однако, отметить, что на основе стандарта IEC 61499 уже было выполнено несколько реальных промышленных проектов [241]. Первая система управления на основе IEC 61499 была внедрена в 2005 г. на мясоперерабатывающем заводе в Новой Зеландии. Австрийская фирма *NxtContol* разработала и внедрила распределенную систему управления зданиями на основе IEC 61499, включающую 2500 датчиков и исполнительных механизмов и 19 контроллеров. В Италии экспериментальная обувная фабрика была оснащена оборудованием, управляемым на основе IEC 61499. Нашел свое применение стандарт IEC 61499 и в авионике. На его основе создана система управления заправкой топливом летательных аппаратов. И, наконец, на основе IEC 61499 построена система управления роботом *Delta* [241].

Интерес к стандарту непрерывно растет как в промышленности, так и в исследовательском секторе. В настоящее время в мире существует несколько компаний и университетов, активно занимающихся созданием соответствующей инфраструктуры для внедрения стандарта IEC 61499 в инженерную практику, к числу которых относятся

фирма *Holobloc Inc*. и Мичиганский университет (США), *Yamatake Corporation* (Япония), Оклендский университет (Новая Зеландия), Венский технологический университет, фирмы *PROFACTOR GmbH* и *nxtControl GmbH* (Австрия), Патрасский университет (Греция), Миланский политехнический университет и Центр исследований и инновационных технологий (Италия), университеты Халле, Магдебурга, Кайзерслаутерна и Ганновера (Германия), Тамперский технологический университет (Финляндия), технологический университет *Chalmers* (Швеция), университет прикладных наук Лугано (Швейцария), фирма *ICSTriplex* и университет Калгари (Канада), Софийский университет химических технологий и металлургии (Болгария), а также ряд других компаний и университетов. Стандарт IEC 61499 был принят сообществом *OOONEIDA* [244], созданным по инициативе *IMS* [161].

С начала исследовательских работ в рамках IEC 61499 было опубликовано три книги, объясняющие идеи IEC 61499 [174, 242, 264]. Как показала практика, этого явно недостаточно. Ситуация будет более контрастной, если взять в качестве образца число книг по объектно ориентированному проектированию. Имеется ряд хороших обзорных статей по стандарту IEC 61499, в которых акцентируется внимание на его преимущества и недостатки, особенности и нерешенные проблемы [226, 241, 263, 265]. В кратком изложении они представлены ниже.

Основными *преимуществами* использования стандарта IEC 61499 являются: повторное использование компонентов, сокращение сроков проектирования, повышение качества и надежности ПО, легкость реконфигурации, наличие предпосылок к проектированию на основе управления моделями (*model driven development*) и архитектурно-центрированного подхода [263].

Особенности IEC 61499:1) дуальность ФБ типа 1, заключающаяся в том, что, с одной стороны, ФБ может быть представлен как процесс, а с другой – как часть кода; 2) дуальность ФБ типа 2, согласно которой ФБ представляет как модель, так и выполнимую спецификацию; 3) выполнение на основе управления событиями (*event-driven execution*); 4) строгая инкапсуляция данных; 5) возможность недетерминированного поведения; 6) открытость входного *XML*-кода [241]; 7) способность к реконфигурации сети ФБ; 8) способность к межузловым взаимодействиям через коммуникационную сеть; 9) дуальность ФБ типа 3: ФБ обладают как свойствами программного, так и аппаратного модуля.

Несмотря на очевидные преимущества стандарта IEC 61499 перед своим предшественником – стандартом IEC 61131-3, его внедрение в промышленную практику идет довольно медленно. Промышленные компании не спешат переходить на новую технологическую базу по ряду объективных причин. Практически во всех обзорах по вопросам, касающимся IEC 61499, отмечаются следующие *проблемы* стандарта IEC 61499, тормозящие его внедрение в производство [226, 241, 263, 265]: 1) неразрешенные семантические проблемы, включающие как неточности в тексте самого стандарта, так и неоднозначность ситуации с моделями выполнения; 2) отсутствие образцовых приложений, которые могли бы служить «примерами для подражания»; 3) отсутствие четких проработанных методологий проектирования; 4) недостаток учебного материала; 5) несовершенство сред разработки и выполнения промышленного масштаба; 6) отсутствие апробированных методов и средств поддержки перехода от проектов стандарта IEC 61131-3 к стандарту IEC 61499. Кроме того, в работе [226] были отмечены дополнительные проблемы: 1) низкоуровневые взаимодействия между ресурсами и устройствами с использованием сервисных интерфейсных функциональных блоков (СИФБ), что увеличивает степень «непрозрачности» между узлами распределенной системы в процессе проектирования; 2) проблема управления качеством обслуживания (*QoS*), что связано с выполнением ограничений реального времени и с надежной коммуникационной инфраструктурой; 3) недостаточность диаграмм (сети ФБ и *ECC*) для того, чтобы описать структуру и поведение управляющего приложения. Как было отмечено в [226], стандарт IEC 61499 не определяет ни пути выявления требований (*requirements elicitation*), ни пути трансформации этих требований в проектные решения. По идее, все это должно решаться использованием эффективного процесса разработки.

1.2. Обзор стандарта IEC 61499

Модель системы

Управляющая система представляется совокупностью устройств, взаимодействующих друг с другом с помощью коммуникационной сети, состоящей из сегментов и линий связи (рис. 1.1).

Функция, выполняемая системой управления, описывается с использованием приложения (*application*), которое может распределяться по одному или нескольким устройствам.

Устройство (*device*) представляет собой физическую сущность, способную выполнять одну или несколько специфицированных функций в определенном контексте и имеющую по меньшей мере один интерфейс, являющийся или интерфейсом управляемого процесса, или коммуникационным интерфейсом [163]. На практике в качестве устройства могут выступать программируемые логические контроллеры (ПЛК), программируемые контроллеры автоматизации (ПКА), промышленные компьютеры. Устройство состоит из одного или нескольких ресурсов.

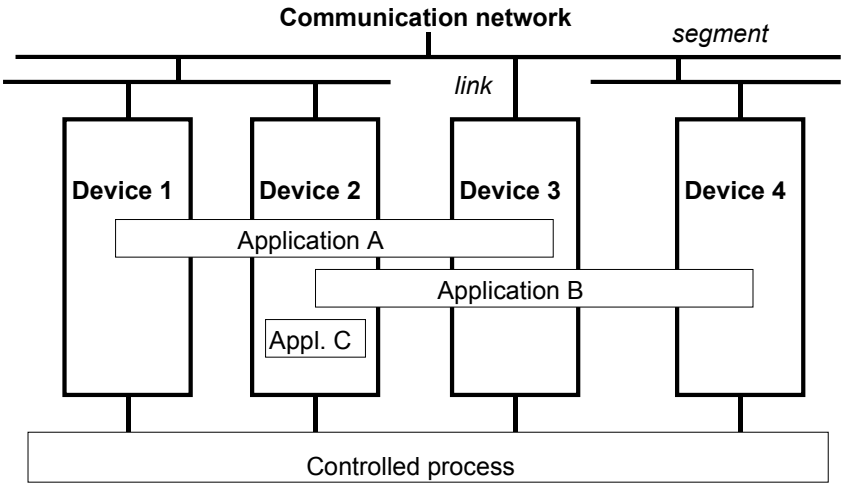


Рис. 1.1. Модель системы IEC 61499 [163]

Ресурс – это функциональная единица, которая имеет независимое управление своими операциями, включая планирование и выполнение алгоритмов [163]. Ресурс может быть создан, конфигурирован, запущен, удален и т.п. без влияния на другие ресурсы. Функциями ресурса являются прием данных и событий из управляемого процесса или коммуникационной сети, их обработка и выдача данных и событий управляемому процессу или коммуникационной сети (как это определено приложениями, выполняемыми на ресурсе). Важнейшей функцией ресурса является функция планирования (диспетчирования), которая определяет порядок выполнения ФБ и передачу данных между ними. Ресурс упрощенно можно представить как небольшую операционную систему в совокупности с исполняемым фрагментом кода.

Приложение является программной функциональной единицей, предназначенной для решения определенной задачи в системе управления. Приложение представляется в виде сети связанных между собой ФБ и субприложений, которые могут выполняться на

различных ресурсах и устройствах системы. Модель приложения представлена на рис. 1.2.

На рис. 1.3 приведен пример соотношения приложений, устройств, ресурсов и сегментов в архитектуре IEC 61499.

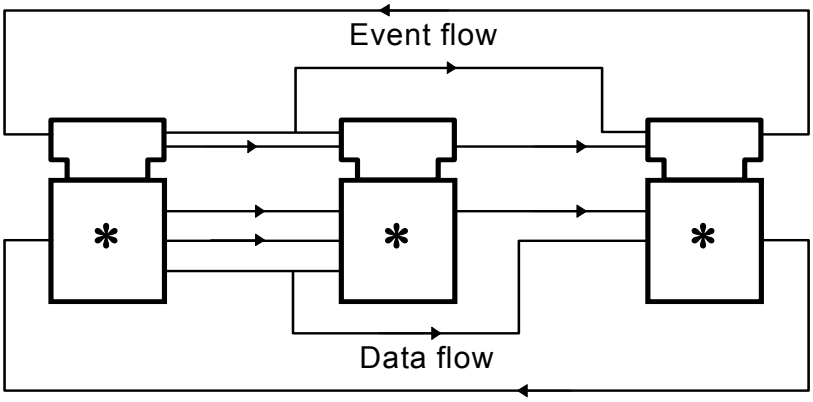


Рис. 1.2. Модель приложения IEC 61499 [163]

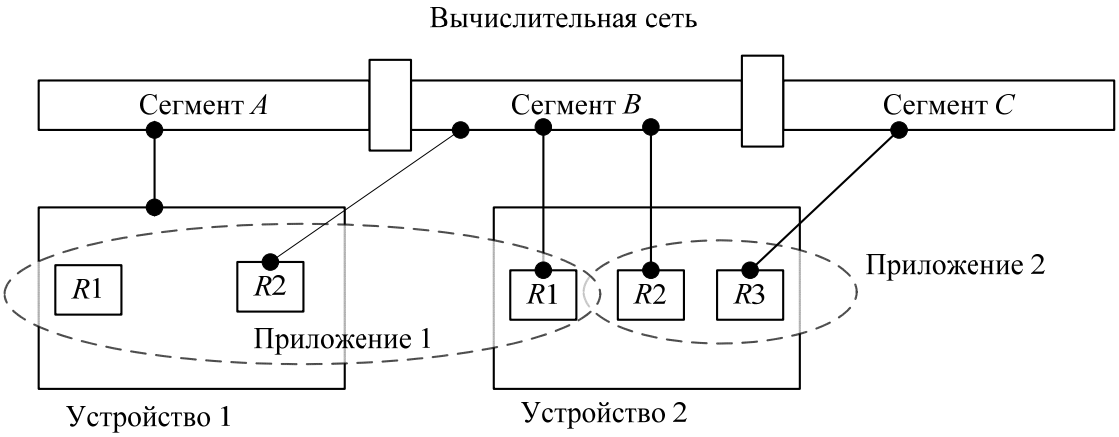


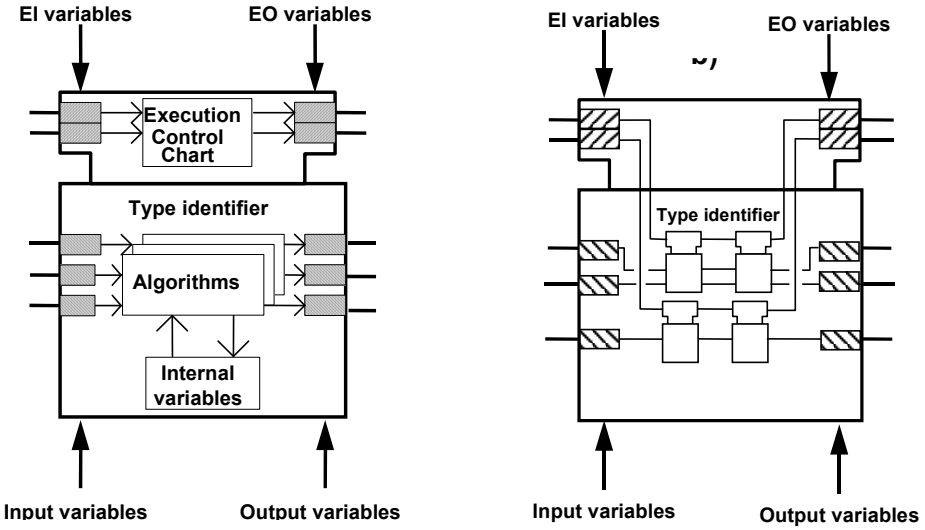
Рис. 1.3. Иллюстрация системных конфигураций

Основным артефактом проектирования в IEC 61499 является функциональный блок (ФБ), представляющий собой специфичный программный компонент, имеющий интерфейс. Интерфейс специфицируется множеством событийных и информационных входов и выходов, через которые соответствующий элемент взаимодействует с окружением. Информационные входы (выходы) определяются в виде входных (выходных) переменных ФБ.

Экземпляры ФБ имеют интерфейсы, во многом сходные с интерфейсами соответствующих типов ФБ, но имеются и существенные отличия. Во-первых, интерфейс экземпляра может быть только подмножеством интерфейса соответствующего типа, некоторые событийные и информационные входы и выходы в интерфейсе экземпляра могут отсутствовать. Во-вторых, в интерфейсе типа ФБ меж-

ду событийными и информационными входами/выходами возможны так называемые *WITH*-ассоциации, которые используются для представления съема данных. В интерфейсах *WITH*-ассоциация представляется в виде вертикального отрезка, связывающего событийный вход (выход) с теми информационными входами (выходами), по которым будет производиться съем (выдача) данных при приходе на событийный вход (выход) сигнала. ФБ управляются с помощью событий, при этом данные являются пассивными, а события (сигналы) – активными.

Существует три вида ФБ: базисный ФБ, составной ФБ и сервисный интерфейсный ФБ (СИФБ). Структуры первых двух типов ФБ приведены на рис. 1.4. Функциональность базисного ФБ определяется машиной состояний (*Execution Control Chart – ECC*) типа автомата Мура (рис. 1.4,*а*). С состояниями *ECC* (ЕС-состояниями) могут быть связаны выходные события и алгоритмы, определенные на одном из языков программируемых контроллеров стандарта IEC 61131-3 [162]. При активизации базисного ФБ некоторым событием он переходит через ряд состояний, выполняя при этом алгоритмы и выдавая выходные сигналы.



а) *б*)

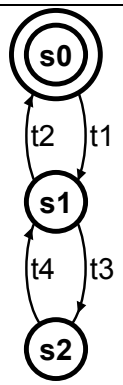
Рис. 1.4. Функциональные блоки стандарта IEC 61499 [163]:
а – базисный ФБ; *б* – составной ФБ

Диаграмма *ECC* выполняется под управлением специальной машины состояний *OSM* (*Operation State Machine*) (рис. 1.5) [163].

Состояние s0 *OSM*-машины можно интерпретировать как свободное состояние ФБ, состояние s1 – как состояние оценки переходов диаграммы *ECC* (ЕС-переходов), а состояние s2 – как состояние выполнения *EC*-акций, связанных с текущим *EC*-состоянием. При-

мер базисного ФБ приведен на рис. 1.6.

Функциональность составного ФБ определяется входящей в него сетью ФБ (рис. 1.7,*б*). С использованием составных ФБ и субприложений производится иерархическая структуризация систем ФБ. Приложение определяется как совокупность ФБ и субприложений, соединенных событийными и информационными связями. Пример составного ФБ представлен на рис. 1.7.



Состояние		**Операции**
s0		
s1		Оценить *EC*-переходы
s2		Выполнить *EC*-акции
Переход	**Условие**	**Операции**
t1	Запустить *ECC*	Съем входных данных
t2	Нет разрешенных *EC*-переходов	
t3	*EC*-переход разрешен	
t4	*EC*-акции завершены	

Рис. 1.5. Диаграмма состояний *OSM* [163]

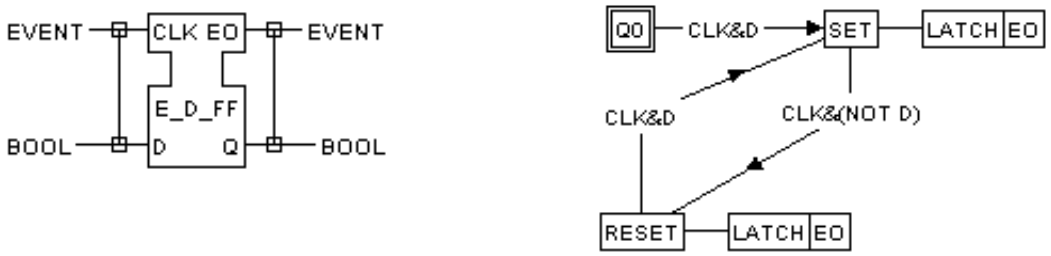


Рис. 1.6. Базисный ФБ «*D*-триггер»:
интерфейс (слева) и диаграмма *ECC* (справа) [163]

Рис. 1.7. Составной ФБ «*D*-триггер»,
срабатывающий по переднему фронту:
интерфейс (слева) и сеть ФБ (справа) [163]

Сервисные интерфейсные функциональные блоки (СИФБ) предоставляют один или несколько сервисов приложению на основе отображения сервисных примитивов на входы и выходы ФБ. СИФБ работают как «обертка», абстрагируя лежащие ниже аппаратные средства. В этом они похожи на драйверы устройств. Сервис предоставляется в виде набора сервисных последовательностей, которые в свою очередь представляются последовательностью сервисных транзакций. Сервисная транзакция, как правило, состоит из входного сервисного примитива и нескольких выходных сервисных примитивов. Последовательность сервисных транзакций определяется в виде диаграмм временных последовательностей (*time-sequence diagrams*) стандарта ISO TR 8509. Время в таких диаграммах увеличивается по направлению вниз. Поэтому сервисные последовательности, а также сервисные транзакции упорядочены по времени, порядок задается их положением в тексте *XML*-описания последовательности. На рис. 1.8 справа в качестве примера приведена диаграмма временных последовательностей, представляющая нормальную передачу данных в клиент-серверной архитектуре [163].

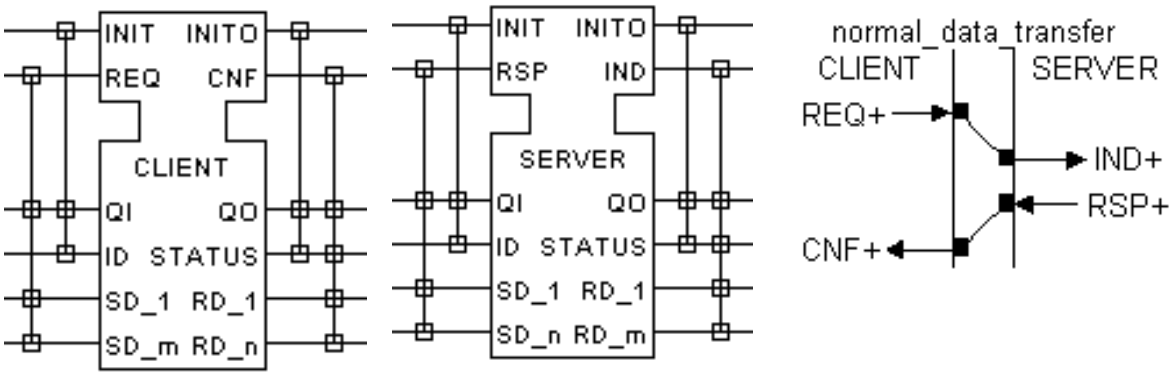
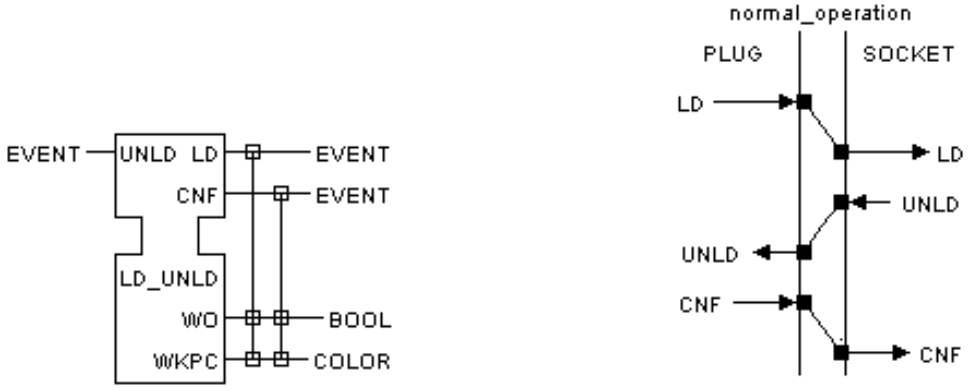


Рис. 1.8. Коммуникационные ФБ (подкласс СИФБ):
интерфейс клиента (слева), интерфейс сервера (в центре),
диаграмма временной последовательности (справа) [163]

Адаптерная связь служит для компактного представления выбранного множества одинарных событийных и информационных связей. Адаптерную связь можно представить как жгут, шину или кабель, а сами адаптеры играют роль разъемов в представлении сложных взаимосвязей между ФБ в системе [163]. Механизм адаптеров позволяет снизить перегруженность схемы соединительными линиями и таким образом масштабировать сложность представляемой схемы. Так же, как и электрические разъемы, различают адаптеры-штекеры (*plug*) и адаптеры-сокеты (*socket*).

Графическое представление типа адаптера приведено на рис. 1.9,*а*. По внешнему виду (имеется ввиду интерфейс) адаптер напоминает ФБ.



а) *б*)

Рис. 1.9. Адаптер:
а – интерфейс; *б* – временная последовательность работы

1.3. Модели выполнения

Стандарт IEC 61499 определяет абстрактную модель ФБ, допускающую различные интерпретации. Подобная недетерминированность модели ФБ и неоднозначность толкования семантики ФБ могут привести к разработке сред выполнения, которые дают разные результаты при интерпретации одной и той же системы ФБ. Все это порождает проблему *портабельности* управляющего ПО, основанного на IEC 61499.

Для того, чтобы поведение приложения (т.е. логической сети ФБ) было полностью определено, необходимо задать так называемую *модель выполнения* ФБ. Назовем моделью выполнения ФБ (*execution model*) набор правил, регламентирующих порядок выполнения сети ФБ на ресурсе и устройстве. Модель выполнения должна по максимуму перевести недетерминированную по своей природе модель ФБ в плоскость детерминизма.

В идеале модель выполнения ФБ должна обладать следующими (возможно противоречащими друг другу) качествами, а именно быть: 1) интуитивно понятной инженеру; 2) простой в реализации; 3) предсказуемой в поведении; 4) справедливой; 5) с хорошей реактивностью; 6) свободной от зацикливаний и оттеснений; 7) верифицируемой; 8) детерминированной; 9) независимой от средств ее реализации (в частности, систем программирования), а также от платформы. Очевидно также и то, что модель выполнения ФБ не должна противоречить модели ФБ, принятой в стандарте IEC 61499, и опираться на нее.

В настоящее время интенсивно ведутся разработки моделей выполнения ФБ. Было предложено несколько моделей выполнения ФБ, в числе которых «непрерываемый многопотоковый ресурс» (*NPMTR*-модель) [219], «прерываемый многопотоковый ресурс» (*PMTR*-модель), последовательные модели, в числе которых модель, основанная на последовательной гипотезе [238, 247], и циклическая модель выполнения [172], синхронные модели выполнения [258], а также модели, реализованные в средах выполнения μ*Crons*, *FUBER* [97] и *CEC*. Некоторые из предложенных моделей опираются на текст стандарта и доопределяют его только в тех вопросах, в которых обнаружены явные лакуны. Другие модели свободно интерпретируют текст стандарта с точки зрения удобства разработчика или ссылаясь на модели реализации. Кроме того, были предложены модели выполнения для некоторых элементов стандарта, а именно: для составных ФБ [218] и базисных ФБ [247]. В работе [49] предлагаются модели выполнения систем интерфейсов ФБ. Таким образом, к настоящему времени накоплен довольно большой объем теоретических и практических знаний в области моделей выполнения ФБ. Каждая из известных моделей выполнения имеет свои преимущества и недостатки и в той или иной мере удовлетворяет многочисленным, порой противоречивым требованиям, предъявляемым к модели выполнения со стороны разработчиков систем управления.

Пробелы в определении семантики ФБ призваны восполнить модели выполнения ФБ, работы по созданию и стандартизации которых были инициированы в 2006 г. сообществом OOONEIDA [188]. К настоящему времени разработан только рабочий проект профиля совместимости [189]. В данном документе предлагаются три основные модели выполнения ФБ: последовательная, циклическая и параллельная. Хороший обзор по существующим моделям выполне-

ния можно найти в [248].

1.3.1. Классификация моделей выполнения ФБ

Классификацию моделей выполнения ФБ можно осуществить различными способами. Например, в [132] модели выполнения были классифицированы по двум признакам: 1) порядок сканирования ФБ (*FB Scan Order*) и 2) многозадачная реализация (*Multitasking Implementation*). Как можно заметить, в данном случае модели выполнения связывались со средой выполнения (что не вполне правильно с точки зрения требований к моделям выполнения). На рис. 1.10 предлагается классификация моделей выполнения ФБ по одному из основных классификационных признаков – по режиму выполнения [47].

Как видно из рис. 1.10, все модели выполнения можно разделить на последовательные и параллельные, последние в свою очередь делятся на синхронные, асинхронные и синхронно-асинхронные. Следует отметить, что 1) существуют и другие классификационные признаки [38]; 2) на рис. 1.10 представлены как классические, так и «нетрадиционные» модели выполнения, обозначенные в [47].

Рис. 1.10. Классификация моделей выполнения ФБ
(по режимам выполнения)

1.3.2. Циклическая модель выполнения

Суть циклической модели выполнения ФБ заключается в том, что ФБ выполняются циклически, последовательно друг за другом

в заранее определенном порядке, например, *fb*1, *fb*4, *fb*2, *fb*3. Таким образом, существует некий список опроса ФБ. Существует несколько вариантов циклической модели, например, в зависимости от того, учитывается ли при запуске ФБ наличие сигналов на его входах. Более подробную информацию о циклической модели можно найти в [172, 248].

1.3.3. Синхронная модель выполнения

Синхронная модель выполнения ФБ предполагает наличие внешних часов. Выполнение гранул (*EC*-переходов или ФБ) производится в моменты дискретного времени …, $t - 1$, t, $t + 1$, … В один момент времени выполняются все гранулы (*EC*-переходы или ФБ), которые могут выполниться.

Синхронная модель выполнения ФБ имеет ряд преимуществ, среди которых: 1) ясность и понятность для инженера; 2) справедливость метода, поскольку раньше выполняются те ФБ, которые получили входной сигнал раньше; 3) отсутствие зацикливаний и оттеснений, возможных в предыдущих моделях.

К недостаткам рассмотренной синхронной модели можно отнести зависимость результата от последовательности выполнения ФБ во фронте. Это касается случая, когда во фронте существуют связанные ФБ. Упорядочение выполнения ФБ во фронте с использованием собственных приоритетов ФБ является в большой степени искусственным приемом, поскольку в общем случае трудно обосновать предпочтение одного ФБ перед другим. Сам стандарт не предусматривает наличия приоритетов у ФБ.

Чтобы избежать зависимости результата от порядка выполнения ФБ во фронте связанных блоков предлагается *двухступенчатая схема* выполнения ФБ с промежуточной буферизацией выходных сигналов [25]. На первой ступени выполняются все ФБ из фронта готовых ФБ, однако передача выходных сигналов ФБ-потребителям с событийных выходов ФБ-производителей откладывается. Можно представить, что данные сигналы на время «замораживаются», т.е. записываются в некоторый промежуточный буфер (модели выполнения). На второй ступени отложенные сигналы «оживают» и доставляются своим потребителям. При двухступенчатой схеме выполнения ФБ порядок выполнения ФБ во фронте становится несущественным за счет того, что при выполнении ФБ учитываются только те входные сигналы, которые были приняты от блоков предыдущего фронта.

Для синхронной модели можно предположить наличие некоторого генератора импульсов, представляющего дискретные часы (рис. 1.11). Каждый импульс будет представлять определенный момент времени.

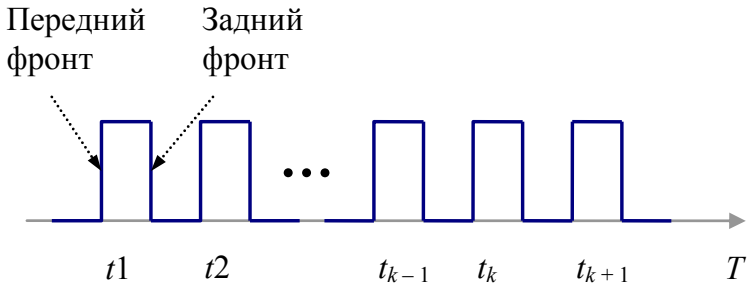


Рис. 1.11. Дискретное время определяется генератором синхроимпульсов

В случае синхронной одноступенчатой модели можно предположить, что все ФБ, входящие во фронт, выполняются по *переднему фронту* синхроимпульса. При двухступенчатой схеме по переднему фронту выполняются действия первой ступени, а по заднему – действия второй ступени.

Более подробную информацию о синхронной модели выполнения можно найти в [34, 51, 248, 258].

1.3.4. Модель выполнения, основанная на последовательной гипотезе

Данная модель выполнения ФБ определена аксиоматически и основана на следующих шести постулатах [238, 247]:

1) функциональный блок (ФБ) может быть активизирован только сигналом на его событийном входе (входным сигналом);

2) выполнение ФБ не может быть прервано выполнением другого ФБ (выполняемого на том же самом ресурсе);

3) выполнение базисного ФБ должно быть «коротким»;

4) входной сигнал ФБ уничтожается после срабатывания одного *EC*-перехода независимо от того, использовался этот сигнал в оценке этого *EC*-перехода или нет;

5) выходные сигналы ФБ выводятся немедленно после того, как соответствующая *EC*-акция завершилась;

6) если ФБ выдает несколько выходных сигналов в одном *EC*-состоянии, то они выводятся последовательно.

При приеме и обработке входного сигнала ФБ выполняется как единое целое (*single run*) до тех пор, пока в нем имеются разре-

шенные *EC*-переходы. Выходные сигналы выводятся последовательно, по мере их возникновения (в хронологическом порядке). Первым будет выполняться тот ФБ, сигнал на который пришел первым, что обеспечивает «справедливость» модели. Для реализации последовательной модели наиболее подходит структура данных типа «Очередь» (или *FIFO*-структура). При решении задач в программировании эта структура данных используется в алгоритмах обхода графа в ширину.

Следует отметить, что убрав шестой постулат, можно получить параллельную модель выполнения [236].

1.3.5. Модель выполнения, основанная на методе «Прямой вызов»

В модели выполнения, основанной на методе «Прямой вызов» (в оригинале – *Direct Call* или *Direct Function Call*), генерация нового события реализуется путем вызова метода *run*, связанного с ФБ [132, 262]. Как отмечено в работе [132], недостатком данного метода является зависимость времени распространения сигнала от топологии сети ФБ, причем это время может быть значительным. На данном методе во многом основывается модель выполнения *NPMTR* [219], реализованная в *FBDK/FBRT* [134]. Отличительная особенность модели – использование вычислений вглубь на уровне *EC*-переходов. Гранулой выполнения является *EC*-акция.

1.3.6. Интегрированная параметризованная модель

В качестве абстрактной модели выполнения ФБ предлагается *интегрированная параметризованная модель выполнения* (ИПМВ) ФБ, в рамках которой предполагается собрать основные полезные свойства и методы существующих моделей, а также добавить ряд новых [38]. Параметризация должна позволить разработчику самому выбирать (конструировать) требуемую модель выполнения путем задания определенных значений параметров (иначе, путем установки режимов). Для разработки ИПМВ использовались морфологические методы. Основой предлагаемой параметризации моделей выполнения ФБ является грануляция вычислений, а также порядок вычисления гранул на основе использования графа гранул.

Ниже предлагается набор морфологических признаков для моделей выполнения ФБ, причем сначала приводится название параметра, а в круглых скобках – его возможные значения: $\boldsymbol{P_1}$ – гранула

выполнения (P_1^1 – функциональный блок; P_1^2 – *EC*-состояние; P_1^3 – *EC*-акция); $\boldsymbol{P_2}$ – порядок вычислений (P_2^1 – обход графа в глубину; P_2^2 – обход графа в ширину, P_2^3 – приоритетный; P_2^4 – циклический); $\boldsymbol{P_3}$ – синхронность выполнения гранул (при P_2^1 и P_2^2)(P_3^1 – последовательное выполнение; P_3^2 – синхронное выполнение); $\boldsymbol{P_4}$ – синхронность передачи сигналов (по нескольким линиям) между парой ФБ (P_4^1 – последовательный; P_4^2 – синхронный); $\boldsymbol{P_5}$ – порядок выдачи сигналов через событийные выходы (при P_4^1)(P_5^1 – естественный, основанный на очередности выдачи выходных сигналов при выполнении инициирующей гранулы; P_5^2 – приоритетный, с учетом приоритетов агрегирующих ФБ для гранул-последователей и приоритетов соответствующих событийных входов, P_5^3 – приоритетный, с учетом приоритета событийного выхода ФБ, P_5^4 – приоритетный, с учетом приоритета событийной связи, соединяющий гранулу-источник с гранулой-приемником); $\boldsymbol{P_6}$ – порядок передач сигналов через один и тот же событийный выход по разным линиям (при P_4^1) (P_6^1 – с учетом приоритетов агрегирующих ФБ и событийных входов для гранул-последователей, P_6^2 – с учетом приоритета событийной связи, исходящей из событийного выхода); $\boldsymbol{P_7}$ – кратность выдаваемых выходных сигналов с событийного выхода (при P_4^1) (P_7^1 – выдается один выходной сигнал; P_7^2 – выдается столько выходных сигналов, сколько их было сгенерировано при выполнении гранулы); $\boldsymbol{P_8}$ – порядок выбора ФБ-последователя (при P_4^2)(P_8^1 – с учетом приоритетов ФБ-последователей; P_8^2 – с учетом максимального приоритета дуги в группе дуг, связывающих ФБ-источник и ФБ-приемник); $\boldsymbol{P_9}$ – порядок выбора входных сигналов при их одновременном приходе на ФБ (при P_4^2)(P_9^1 – определяется приоритетом событийного входа; P_9^2 – определяется приоритетом входной событийной дуги; P_9^3 – определяется естественным порядком выдачи сигналов с ФБ-источников); $\boldsymbol{P_{10}}$ – метод выполнения составных ФБ (P_{10}^1 – как единой сущности; P_{10}^2 – как контейнера); $\boldsymbol{P_{11}}$ – приоритетность базисных ФБ перед составными ФБ (при P_{10}^1)(P_{11}^1 – больше; P_{11}^2 – меньше, P_{11}^3 – равноценны); $\boldsymbol{P_{12}}$ – степень интегрированности модели выполнения сети клапанов данных (КД) в модель выполнения системы ФБ (при P_{10}^2) (P_{12}^1 – сеть КД выполняется как единое целое отдельно от системы ФБ и взаимодействует с общей моделью, когда завершает свою работу; P_{12}^2 – система ФБ и сеть КД выполняются совместно, как единое целое); $\boldsymbol{P_{13}}$ – приоритетность сети КД перед базисными ФБ (при P_{12}^1)(P_{13}^1 – больше; P_{13}^2 – меньше, P_{13}^3 – равноценны); $\boldsymbol{P_{14}}$ – метод выполнения сети КД (при P_{12}^1)(P_{14}^1 – последовательное выполнение; P_{14}^2 – синхронное выпол-

нение; P_{14}^{3} – последовательно-синхронное выполнение); $\boldsymbol{P_{15}}$ – тип *EC*-переходов без событий в диаграмме *ECC* базисного ФБ (P_{15}^{1} – пассивные, P_{15}^{2} – активные); $\boldsymbol{P_{16}}$ – тип выполнения транзакции по обработке внешнего сигнала (P_{16}^{1} – непрерываемый, P_{16}^{2} – прерываемый, P_{16}^{3} – смешанный); $\boldsymbol{P_{17}}$ – способ диспетчеризации внешних сигналов (при P_{16}^{1})(P_{17}^{1} – естественный, с учетом времени прихода внешнего сигнала, с сохранением; P_{17}^{2} – приоритетный, с учетом приоритета внешнего сигнала, с сохранением; P_{17}^{3} – немедленное восприятие, с потерей). Следует отметить, что не все значения параметров совместимы между собой.

Более подробную информацию о ИПМВ ФБ можно найти в [23, 38].

1.3.7. Модели выполнения на основе спусковых функций

Модели выполнения данного класса строятся на основе механизма *спусковых функций* [47]. С его помощью можно однозначно задать порядок выполнения ФБ. Спусковые функции строятся с использованием предикатов и функций *времени выполнения* ФБ. Строго говоря, модель выполнения, кроме порядка выполнения, включает правила обработки входных сигналов и выдачи выходных сигналов, правила передачи сигналов и данных между блоками, правила интерпретации диаграммы *ECC*, но среди данных аспектов порядок выполнения ФБ является определяющим.

Для пояснения модели предварительно введем ряд определений.

$FB = \{fb_1, fb_2, \ldots, fb_n\}$ – множество ФБ, входящих в систему и подлежащих исполнению.

$EI^j = \{ei_1^j, ei_2^j, \ldots, ei_k^j\}$ – множество событийных входов блока fb_j. В дальнейшем для простоты верхний индекс будет опускаться.

$ZEI: EI \rightarrow \{0,1\}$ – функция значений входных событийных переменных. Если $EI(ei_j) = 1$, то входная событийная переменная ei_j установлена (иными словами, сигнал присутствует на входе), иначе – сброшена.

Предикат $activeFB: FB \rightarrow \{true, false\}$ определяет ФБ, являющиеся *активными* в текущий момент модельного времени. Предикат задает так называемую функцию возбуждения (или *спусковую функцию*). Интерпретатором будет сделана попытка запуска активного ФБ. В дальнейшем для краткости предикаты будем определять только именем, без показа самого отображения. Пример опре-

деления вышеприведенного предиката: $activeFB(fb_i)$. Обозначим $FB_a = \{fb \in FB \mid activeFB(fb)\}$ – множество активных ФБ в текущий момент модельного времени.

Предикат $activeFBPrev(fb_j)$ определяет, являлся ли блок fb_j активным в предыдущий момент модельного времени. Обозначим $FB_{ap} = \{fb \in FB \mid activeFBPrev(fb)\}$ – множество ФБ, которые были активными непосредственно в прошлом.

Предикат $enabled(fb_j) = \exists\, ei \in EI\ [ZEI(ei) = 1]$ определяет, является ли разрешенным в текущий момент модельного времени блок fb_j. ФБ является *разрешенным*, если хотя бы на одном из его событийных входов находится сигнал. Определим $FB_e = \{fb \in FB \mid enabled(fb)\}$ – множество разрешенных ФБ.

Функция $EOS(fb_j)$ определяет линейно-упорядоченное мультимножество сигналов (последовательность), выданных блоком $fb_j \in FB$ при своем выполнении. В данном случае сигналы идентифицируются событийными выходами ФБ, на которых они находятся. Упорядоченность определяется порядком выдачи сигналов.

Функция $EIS(fb_j)$ определяет последовательность сигналов, которые появились на входах ФБ в результате выполнения блока fb_j. Упорядоченность определяется порядком появления сигналов на входах.

Функция $OF(fb_j)$ определяет последовательность блоков-приемников сигналов, выданных с блока fb_j при его выполнении. Упорядоченность определяется порядком получения соответствующими блоками входных сигналов.

Функция pr: $FB \cup EI \rightarrow N_0 = \{0, 1, 2, \ldots\}$ определяет приоритет блока или событийного входа.

Отношение предпочтения $Pref \subseteq FB \times FB$ служит для определения наиболее предпочтительных блоков для выполнения. Это отношение частичного порядка. Если $(fb_i, fb_j) \in Pref$, то выполнение блока fb_i предпочтительнее выполнения блока fb_j.

Отношение $PoolOrder \subseteq FB \times FB$ задает жесткий порядок выполнения ФБ в циклической модели. С помощью этого отношения все ФБ связаны в кольцо опроса.

Отношение $EvConn \subseteq FB \times FB$ определяет событийные связи между ФБ как целыми единицами. Обозначим pre_{fb} множество предшественников блока fb по отношению $EvConn$.

Предикат $activeEI(ei_j)$ определяет, является ли событийный вход ei_j активным. Сигнал на активном входе подлежит обязательной об-

работке.

Функция τ^j: $EI^j \to N \cup \{\omega\}$ определяет времена прибытия (порядковые номера) сигналов на событийных входах блока fb_j. В дальнейшем для простоты верхний индекс будет опускаться. Если $\tau(ei_k)= \omega$, то считается, что на событийной линии ei_k сигнал отсутствует. В каждый новый момент модельного времени локальные часы для регистрации входных сигналов в системе сбрасываются в ноль.

С помощью спусковых функций можно определить практически любую модель выполнения. Примеры определения «классических» моделей выполнения представлены ниже.

Циклическую модель выполнения можно определить следующей функцией возбуждения:

$$activeFB(fb_j) = \exists\, fb_i \in FB[activeFBPrev(fb_i)\ \&\ (fb_i, fb_j) \in PoolOrder]$$

Циклическая модель выполнения с пропуском неразрешенных ФБ определяется следующим образом:

$$activeFB(fb_j) = enabled(fb_j)\ \&\ (\exists\, fb_i \in FB_{ap}[\exists s=(fb_i, fb_{i+1}, \ldots, fb_j)$$
$$[\forall k \in \{i, i+1, \ldots, j-1\}[(f_k, f_{k+1}) \in PoolOrder]\ \&\ |s|>2 \to$$
$$\to \forall m \in \{i+1, \ldots, j-1\}[\neg enabled(fb_j)]] \vee$$
$$\bar{\exists}\, fb_i \in FB_{ap}\ \&\ \exists\, fb_i \in FB_e[pr(fb_i) > pr(fb_j)]).$$

Модель выполнения на основе последовательной гипотезы может быть определена следующим образом:

$$activeFB(fb_j) = \exists\, fb_i \in prefb_j[fb_j \in OF(fb_i)\ \&\ (\exists fb_k \in OF(fb_i)$$
$$[activeFBPrev(fb_k)\ \&\ fb_j = next(fb_k)] \vee activeFBPrev(fb_i)\ \&\ first(OF(fb_i), fb_j) = true)].$$

В классической синхронной однотактной модели в каждый момент модельного времени выполняются все разрешенные ФБ:

$$activeFB(fb_j) = enabled(fb_j).$$

В синхронной модели с использованием частичного порядка срабатывания в каждый момент модельного времени активизируются только наиболее предпочтительные разрешенные ФБ:

$$activeFB(fb_j) = enabled(fb_j)\ \&\ \bar{\exists}\, fb_i \in FB[enabled(fb_i)\ \&\ (fb_i, fb_j) \in \in Pref].$$

Рассмотрим следующий пример. Пусть отношение предпочтения *Pref* задано в виде диаграммы Хассе на рис. 1.12. Тогда, если допустить, что разрешены все ФБ *fb*1…*fb*8, будет активизирован только один блок *fb*1 как наиболее предпочтительный. Если же разрешены все ФБ, кроме *fb*1, то одновременно будут активизированы блоки *fb*2, *fb*3 и *fb*4.

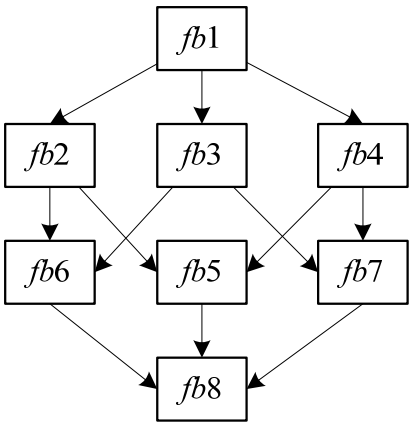


Рис. 1.12. Диаграмма Хассе,
определяющая отношение предпочтения ФБ

Синхронную модель с использованием приоритетных уровней можно считать модификацией предыдущей модели. В ней одновременно активизируются только наиболее приоритетные разрешенные ФБ:

$activeFB(fb_j) = enabled(fb_j)\ \&\ \neg\exists\, fb_i \in FB[enabled(fb_i)\ \&\ pr(fb_i) > pr(fb_j)]$.

В качестве примера рассмотрим реализацию синхронной модели на основе частичного порядка с помощью логических выражений. В функцию возбуждения включается конъюнкция инверсных значений предикатов разрешенности тех ФБ, которые предпочтительнее данного ФБ. Для блока *fb*5, приведенного в примере, представленном на рис. 1.12, можно определить следующую функцию возбуждения:

$activeFB(fb5)=enabled(fb5)\ \&\ \neg enabled(fb1)\ \&\ \neg enabled(fb2)\ \&\ \neg enabled(fb4)$.

С использованием механизма спусковых функций можно задать «нетрадиционные» модели выполнения, например, основанные на групповых взаимодействиях. Синхронная модель выполнения с явным группированием предполагает разбиение ФБ на группы $G_1, G_2, \ldots, G_n$ и активизацию блоков целыми группами. Возможно вхождение одного ФБ в разные группы, т.е. возможно, что $G_i \cap G_j \neq \varnothing$. В группе ФБ может выделяться один или несколько лидеров группы. Можно выделить следующие условия разрешенности группы: а) разрешены все члены группы; б) разрешен один из лидеров группы; в) разрешены все лидеры группы; г) условие в виде произвольного логического выражения. Одновременно может быть разрешено несколько групп, что определяет конфликтную ситуацию. Возможны следующие пути разрешения конфликтов: а) на основе приори-

тетов групп; б) на основе максимального числа разрешенных ФБ в группах; в) на основе максимального числа разрешенных ФБ-лидеров в группах и т.д.

1.3.8. Обработка входных сигналов

Обработка входных сигналов (в базисных ФБ) является одним из самых дискуссионных и «скользких» моментов в моделях выполнения ФБ. Это связано прежде всего с тем, что сам стандарт не определяет полно жизненный цикл входных сигналов, это особенно касается тех моментов, которые связаны с их удалением. Попробуем еще раз выделить типовые ситуации, возникающие на событийных входах базисных ФБ, а также пути их решения [47, 119].

«Хорошая» ситуация, однозначно интерпретируемая стандартом IEC 61499 и не вызывающая проблем в моделях выполнения ФБ, приведена на рис. 1.13. Это случай, когда на один из событийных входов поступает сигнал, и при этом существует разрешенный переход в диаграмме *ECC*, помеченный этим сигналом.

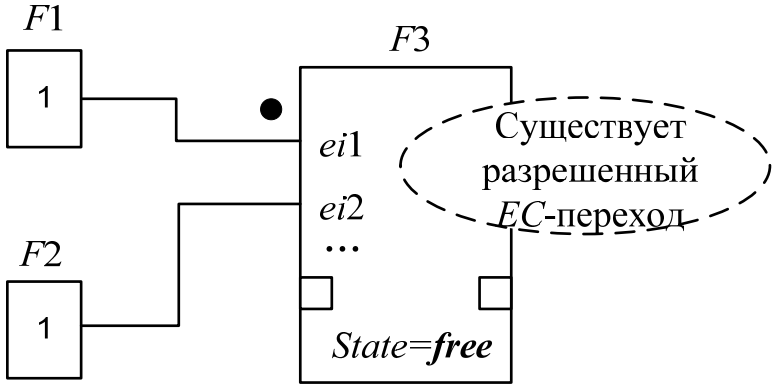


Рис. 1.13. Однозначная ситуация с входными сигналами

Если предположить параллельное асинхронное функционирование ФБ, которое покрывает все возможные сценарии развития событий в системе ФБ, то, кроме «хорошей» ситуации, представленной на рис. 1.13, возможны и другие ситуации, которые, однако, вызывают различные трактовки у разных исследователей. Назовем эти ситуации проблемными. Схематичное представление подобных ситуаций приведено на рис. 1.14. На данном рисунке состояние операционного автомата (*OSM*) обозначено как *State*. Значение *free* («свободно») соответствует состоянию *s*0 автомата *OSM*, а состояние *busy* («занято») – состояниям *s*1 или *s*2 автомата *OSM* [162].

Ситуация 1 (рис. 1.14,*а*), когда на входы ФБ одновременно приходят несколько сигналов, является маловероятной, но возможной. Более вероятной является ситуация 2 (рис. 1.14,*б*), когда в занятом

состоянии на ФБ приходит сигнал. И чем больше реальная продолжительность выполнения алгоритмов, тем более вероятна данная ситуация. Ситуацию 3 можно считать типовой и она встречается на практике очень часто. Не рассматриваются отдельно ситуации 2 и 3 в случае прихода нескольких сигналов, поскольку это не меняет сути дела. Также особо не рассматривается ситуация, когда на один событийный вход приходит два (или более) сигнала с разных направлений, поскольку событийный вход с несколькими входящими событийными линиями можно заменить на эквивалентную конструкцию, содержащую стандартный ФБ *E_MERGE*. В этом случае данная ситуация трансформируется в ситуацию 1. Ситуация, когда на один и тот же событийный вход одновременно приходят два (или более) различных сигнала с одной и той же событийной линии, является логически бессмысленной. В этом случае «серия» из нескольких сигналов может быть представлена одним сигналом (хотя возможны другие варианты).

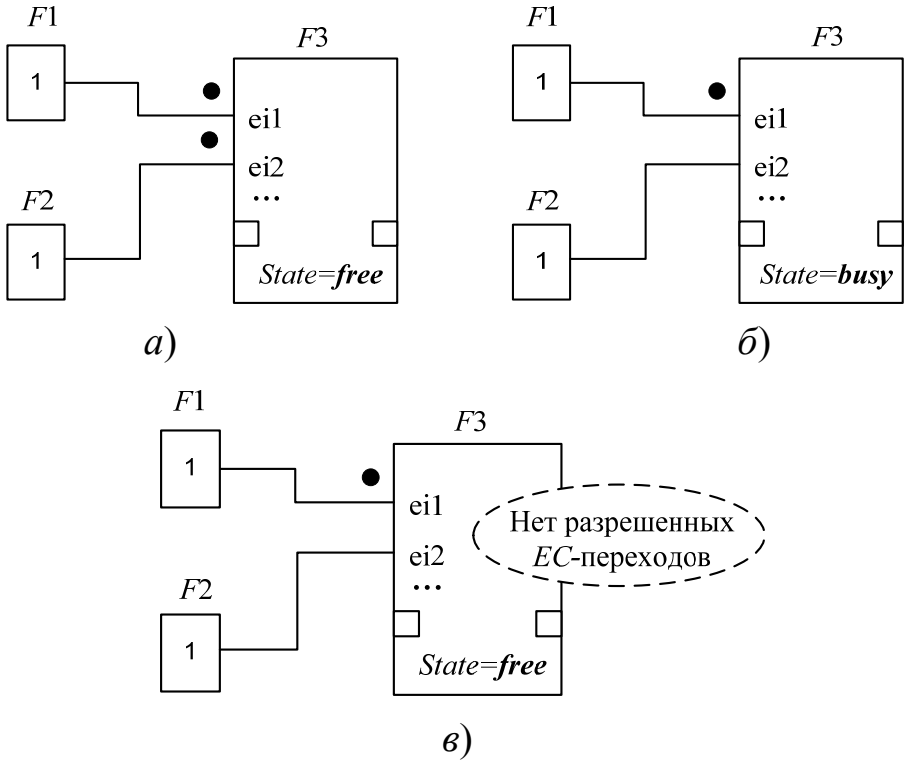


Рис. 1.14. Проблемные ситуации с входными сигналами:
а – «Одновременный приход сигналов на входы ФБ» (ситуация 1);
б – «Приход сигнала на занятый ФБ» (ситуация 2);
в – «Приход сигнала на незанятый ФБ, когда его обработка не предусмотрена в *ECC*» (ситуация 3)

Для обеспечения детерминизма поведения системы ФБ модель выполнения призвана поддерживать только один из сценариев развития событий в системе. Исходя из фактора возможности про-

блемных ситуаций, приведенных выше, их следует рассматривать (а не игнорировать рассмотрение) в моделях выполнения ФБ, в которых они возникают. Следует, однако, заметить, что не все модели выполнения предполагают наличие всех перечисленных ситуаций. Например, в модели выполнения на основе последовательной гипотезы в принципе невозможна ситуация 1.

Рассмотрим возможные пути решения проблемных ситуаций. По ситуации 1 имеются следующие варианты обработки входных событий ФБ:

1) по какому-либо правилу выбирается один (активный) входной сигнал, который будет в дальнейшем обрабатываться, а другие сигналы удаляются (правило DelEI1):

$$\forall ei \in EI \setminus ei_a \ \{ZEI(ei) = 0\}.$$

Здесь под ei_a обозначен событийный вход, на котором находится активный сигнал, а в фигурные скобки заключены исполняемые действия. Потерю сигнала можно считать недостатком этого правила, поскольку, по сути дела, теряется (возможно, важная) информация;

2) обрабатываются все входные сигналы, но по одному, в порядке предпочтительности. Фаза активности ФБ включает обработку всех входных сигналов последовательно, один за другим (правило DelEI2);

3) обрабатывается один (активный) сигнал, выбираемый по определенному правилу, а остальные не трогаются (сохраняются). После обработки этого сигнала могут выполняться другие разрешенные ФБ. Фаза активности ФБ включает обработку одного сигнала (правило DelEI3).

Возможны следующие варианты выбора активного сигнала:

1) по приоритету входной событийной линии. При этом предикат выбора определяется как

$$activeEI(ei_j) = ZEI(ei_j) = 1 \ \& \ \overline{\exists} \ ei_k \in EI \ [pr(ei_k) > pr(ei_j)];$$

2) по времени поступления сигнала:

$$activeEI(ei_j) = ZEI(ei_j) = 1 \ \& \ \overline{\exists} \ ei_k \in EI \ [\tau(ei_k) < \tau(ei_j)];$$

3) по наличию и объему действий, связанных с входным сигналом. При этом в первую очередь обрабатываются сигналы, связанные с *EC*-переходами, выходящими из текущего состояния, поскольку эти сигналы могут вызвать самые крупные активности в работе ФБ – переходы из одного состояния *ECC* в другое, а также связанные с этим возможные *EC*-акции. Во вторую очередь обрабаты-

ваются сигналы, связанные только со съемом данных. Остальные сигналы заведомо являются «пустыми» в плане связанных с ними действий. Следует отметить, что перечисленные выше три правила выбора могут комбинироваться.

Для решения ситуации 2 возможны следующие варианты обработки входного сигнала: 1) сигнал на входе уничтожается; 2) сигнал на входе сохраняется. Можно предполагать, что стандарт скорее склоняется к варианту 1, в то время как черновик стандарта был ориентирован на вариант 2, поскольку в нем вводились очереди сигналов единичной длины (так называемые *EI*-переменные). Ситуация 3 может решаться теми же двумя способами, что и ситуация 2, т.е. сигнал на входе или уничтожается, или нет. Скорее всего в стандарте предполагался первый вариант. Основанием для этого является Таблица 1 стандарта IEC 61499 [163].

1.4. Формализованное описание и верификация систем управления на основе IEC 61499

Формализация объекта проектирования и самого процесса проектирования имеет исключительно важное значение для создания надежного управляющего ПО, удовлетворяющего потребителей как по функциональным и временным параметрам, так и по его качественным характеристикам, в заданные (как правило, сжатые) сроки. Основными причинами для формализации систем управления на основе IEC 61499 являются: 1) определение семантики; 2) верификация; 3) оценка производительности; 4) эквивалентные преобразования или трансформация в другие целевые модели (задача синтеза).

1.4.1. Обзор работ по формальным моделям

Исторически первыми для моделирования систем ФБ IEC 61499 использовались модульные *NCES*-сети [203], берущие свое начало от сетей Петри [59, 65]. Первым исследованием в этой области была работа [235]. К преимуществам *NCES*-сетей относятся: 1) возможность моделирования синхронно-асинхронных систем, а также импульсных сигналов и сигналов в виде потенциала; 2) наличие средств иерархической структуризации и модульности; 3) простота и большая вычислительная мощность; 4) наличие инструментальных средств анализа. В работе [216] показано, что *NCES*-сети равномощны машинам Тьюринга, что делает их универсальным средством моделирования систем ФБ любой степени сложности.

В работах [137, 197] освещаются вопросы моделирования алгоритмов ФБ с помощью *NCES* на уровне целочисленной арифметики.

В [239] *NCES*-сети использовались для представления моделей выполнения ФБ. В последнее время были предприняты новые попытки автоматизации моделирования систем ФБ на основе *NCES*-сетей. В работе [175] были предложены неформальные правила для создания моделей некоторых элементов ФБ, а также моделей их выполнения. Данные правила были реализованы в системе *FBSD*, разработанной в среде *Eclipse*. В работе [166] правила преобразования реализуются в виде *XSLT*-скриптов и правил языка Пролог. Следует отметить, что работы [166, 175] ограничены в плане моделирования, например, в них не учитывается диаграмма состояний операций *ECC* (см. *Figure* 12 в [163]). В работе [237] разрабатывается концепция интеллектуальных мехатронных компонентов для возможности проектирования формальных моделей (на основе *NCES*), используемых в процессе формальной верификации и обозначающих путь для интеграции имитационного моделирования и формальной верификации. В работе [169] *NCES*-сети использовались для моделирования и верификации протокола реконфигурации для мультиагентного управления в архитектуре IEC 61499. Следует отметить, что, несмотря на значительный прогресс в области моделирования и автоматизации построения формальных моделей ФБ на основе *NCES*-сетей, можно выделить следующие основные нерешенные проблемы: 1) сложность моделирования потоков данных и алгоритмов; 2) отсутствие формальных правил преобразования систем ФБ в *NCES*-сети; 3) сложность перепрограммирования систем синтеза при изменении правил. Для решения первой проблемы в [28] предложены арифметические *NCES*-сети, являющиеся расширением классических *NCES*-сетей. Эти сети расширены функциями по обработке данных и логическими условиями переходов.

К одной из первых работ, посвященных отмеченной теме формализации, можно отнести также работу [125]. В ней предложен формальный язык функциональных блоков *FBComposed*, основанный на использовании шаблонов для построения модульных систем ФБ. При разработке шаблонов использовались положения стандарта IEC 61499 и доменно-ориентированные расширения, облегчающие разработку алгоритмов обработки в системах ФБ. Предлагается использование дискретных, непрерывных и гибридных модулей, замещающих шаблоны. Язык определяется с помощью системы перезаписи графов *PROGRES* [206]. Ограниченность подхода определяется готовыми шаблонами и модулями, имеющимися в наличии. Предложенный в работе подход носит декларативный характер и не подкреплен конкретными примерами построения моделей по опи-

саниям систем ФБ.

Кроме *NCES*-сетей, для формального представления ФБ использовалось другое расширение сетей Петри – сигнально-интерпретированные сети Петри (*SIPN*) [170]. В работе [148] для описания систем ФБ были предложены *XNet*-сети, основными элементами которых являются события, ресурсы и обработчики.

Второй по популярности подход к формальному описанию ФБ (после расширенных сетей Петри) – использование конечно-автоматных моделей. В работе [96] для моделирования поведения базисных ФБ использовались расширенные конечные автоматы, в [46] – временные автоматы, в [78] – модульные конечные автоматы типа «событие–условие–действие» (*ECA MFSM*) и сети Петри.

Модель переходов состояний, разработанная специально для формального описания систем ФБ, представлена в [48, 117, 120]. В том же направлении можно считать работы [98, 99]. Эти работы продолжили работы [48, 117, 120] в плане расширения моделей дисциплинами выполнения ФБ.

1.4.2. Проблема определения формальной семантики ФБ

Формальная семантика является областью знаний, имеющей отношение к строгому математическому исследованию смысла языков программирования и моделей вычислений [58, 146, 253]. Формальная семантика языка задается математической моделью, которая описывает возможные вычисления, задаваемые языком.

Существующее положение дел в этой сфере в отношении IEC 61499 характеризуется следующими факторами: 1) наличием пробелов в определении семантики ФБ в самом тексте стандарта IEC 61499, внесенных (по одной из версий) сознательно, чтобы не ограничивать свободу в развитии концепции ФБ; 2) наличием большого числа моделей выполнения ФБ; 3) отсутствием общепризнанной формальной семантики ФБ.

Отсутствие целостного определения формальной семантики ФБ порождает целый ряд проблем: 1) непонимание детальной работы ФБ приводит к различным интерпретациям ФБ и различным реализациям программно-аппаратных платформ, что в свою очередь может привести к проблеме портабельности управляющего ПО; 2) проведение верификации и имитационного моделирования на основе упрощенных (и, возможно, ошибочных) моделей может привести к неправильным результатам, что, в свою очередь, напрямую отражается на надежности управляющего ПО; 3) рост скептицизма по отношению к стандарту IEC 61499 и замедление темпов его вне-

дрения в промышленную практику. Практически полное отсутствие работ по определению формальной семантики ФБ определяется следующими причинами: 1) кажущейся простотой конечно-автоматной модели, лежащей в основе IEC 61499; 2) неопределенностью с моделями выполнения ФБ (в плане стандартизации); 3) недостаточной востребованностью стандарта IEC 61499 в промышленности. В определенной степени изложенные проблемы (что касается пункта 3) образуют замкнутый круг.

Сложность ФБ в семантическом плане определяется следующими факторами: 1) наличием различных, порой нетривиальных, моделей выполнения; 2) наличием алгоритмов обработки данных в базисных ФБ; 3) выполнением ФБ на основе управления событиями (*event driven execution*); 4) иерархичностью и модульностью системы ФБ; 5) наличием типизации и инстанциации большинства артефактов проектирования (ФБ, субприложений, ресурсов, устройств); 6) наличием у ФБ свойств как программного модуля, так и аппаратного; 7) язык ФБ является *визуальным* языком программирования, использующим блоковые диаграммы с встроенной машиной состояний, неограниченным иерархическим вложением блоков друг в друга и их распределением по сетевым устройствам, использованием принципа потока событий как основы вычислений. Подобные визуальные компонентно-базированные событийно ориентированные языки требуют особых подходов к описанию их семантики.

Кратко рассмотрим работы, в определенной степени связанные с формальной семантикой ФБ. Модели ФБ на основе *NCES*-сетей, конечных и временных автоматов, сетей Петри, рассмотренные в предыдущем подразделе, нельзя отнести к формальной семантике ввиду их абстрактности. Например, в них не нашли достаточное отражение все элементы языка ФБ. В работах [48, 117] семантика ФБ представлена в виде системы переходов состояний, но не в полной мере (без учета конкретных моделей выполнения) и местами неформально. В [257] разработана структурная операционная семантика ФБ, но только для синхронной модели выполнения. В работах [49, 50] предложена формальная рамочная нотация для моделирования ФБ в различных моделях выполнения и рассмотрены три модели выполнения. В принципе данную модель можно считать формальной семантической моделью, но с некоторыми оговорками: 1) не рассматривается *OSM*-машина, играющая ключевую роль в выполнении базисного ФБ и определенная в стандарте IEC 61499; 2) предложенная модель является довольно абстрактной в ряде по-

зиций, например, алгоритм рассматривается как некая абстрактная функция, без детализации шагов его выполнения, что не является критичным, но важно, например, при аппаратной реализации ФБ. Как следствие этого, описание системы ФБ в соответствии с предложенной формальной моделью нельзя считать выполнимой спецификацией; 3) предложенная формальная модель не вполне четко позиционирована среди известных формализмов. Можно предположить, что это некая система переходов состояний. В то же время использование явных алгоритмов при описании выводит ее в плоскость полуформальных моделей.

Ниже производится обоснование выбора наиболее приемлемой формальной нотации для представления семантики ФБ. Как известно, существует множество подходов к определению формальной семантики, но основными являются следующие три класса формальных семантик: денотационная, операционная и аксиоматическая [58, 146, 253]. Кроме этих видов, также часто используются алгебраические спецификации, машины абстрактных состояний, монадическая семантика, семантика действий [183]. Выбор формализма для описания семантики языка зависит как от класса языка и особенностей, так и от необходимости дальнейшего анализа программ. Существует закономерность, согласно которой при увеличении вычислительной мощности формализма увеличивается сложность его анализа и верификации. Здесь уместен пример с введением в сеть Петри ингибиторных дуг, что позволило сетевым моделям достичь моделирующей мощности машин Тьюринга, но в то же время исключило из арсенала инструментов исследования матричный анализ.

Операционная семантика языков программирования описывает, как правильные программы интерпретируются в виде последовательности шагов вычислений [142, 154]. Эта семантика тесно связана с интерпретацией. Операционная семантика может определять некую абстрактную машину и задавать значение фраз языка путем описания переходов, которые эти фразы вызывают на состояниях машины. Очевидно, что операционная семантика является не чем иным, как формализованным описанием содержательной семантики (которую можно, таким образом, назвать интуитивной операционной семантикой). Наибольшую популярность для описания семантик языков программирования получила так называемая структурная операционная семантика (*SOS*), использующая в качестве основы правила вывода [142].

В последнее время все большее распространение для определения операционной семантики языков находит аппарат машин абстрактных состояний (МАС), введенный Ю. Гуревичем [83, 147]. Эта формальная нотация имеет ряд преимуществ перед другими методами определения формальной семантики: 1) выразительность, простота и понятность; 2) возможность представления алгоритмов на различных уровнях – от абстрактного до уровня реализации; 3) формальное описание может использоваться как выполнимая спецификация, что обеспечивает легкость создания различных сред выполнения.

МАС обобщают хорошо известные машины конечных состояний. В качестве состояния в МАС используется структура над заданным словарем γ (иначе, алгебра), включающая базовое множество (или набор базовых множеств) и сигнатуру – набор функций (называемых операциями), заданных на этом множестве. Для представления динамики моделируемой системы используется механизм обновления функций. Обновления специфицируются в терминах операторов обновления в форме $f(t_1,\ldots,t_n) := t_0$, где левая часть – терм над γ, идентифицирующий локацию («точку», где надо изменить значение функции), здесь f – это n-местная функция из сигнатуры, а $t_1, \ldots, t_n$ – термы; а правая часть (терм) – новое значение функции. Сложные правила переходов строятся из операторов обновления рекурсивно с помощью определенных конструкторов правил. В упрощенной форме правило обновления функции имеет вид

ЕСЛИ <*Условие*>, *ТО* <*Обновления*>,

где <*Условие*> специфицирует сторожевое условие в виде формулы логики первого порядка (булевское выражение), а <*Обновления*> – это набор операторов обновления. По сути дела, правила определяют переходы в пространстве состояний МАС. Множество правил образуют программу ρ для МАС. Вычисления в МАС производятся по шагам. Шаг вычисления состоит из следующих операций, выполняемых как неделимое действие: 1) вычисление множества обновлений Δ_S путем оценки правил из программы ρ; 2) проверка Δ_S на консистентность; 3) срабатывание Δ_S в состоянии S. Прогон машины μ – это последовательность состояний S_0, S_1, S_2, …, такая, что S_{i+1} достижимо из S_i при срабатывании Δ_{Si} в состоянии S_i. Следует отметить, что все (консистентные) обновления в классических МАС выполняются одновременно [223].

К настоящему времени предложено множество вариантов МАС, среди которых недетерминированные, параллельные, синхронные,

асинхронные, мультиагентные, распределенные и интерактивные МАС [83, 95, 140, 147]. В некоторых работах определяются сходные модели, но при этом используется разная терминология и нотация.

Мультиагентные МАС могут быть синхронными и асинхронными [83, 95, 140, 147]. Мультиагентные *синхронные* МАС определяются как множество агентов, которые выполняют собственные МАС параллельно, синхронизируясь, используя неявные глобальные часы. Семантически синхронные МАС эквивалентны множеству составляющих одноагентных МАС, работающих на глобальном состоянии через объединение сигнатур компонентных МАС. Они выполняются квазипоследовательно.

Мультиагентные *асинхронные* МАС (иногда называемые *распределенными* МАС) задаются множеством пар, связывающих агентов с компонентными МАС. Выполнение асинхронных МАС частично упорядочено, причем при этом должны выполняться три условия: а) конечность истории; б) последовательность агентов; в) когерентность (согласованность) [147]. Внутри шага вычислений (в соответствии с некоторым недетерминированным выбором) один или более агентов активизируют свои действия по обновлению одновременно. Семантика модели гарантирует (путем ограничения класса допустимых прогонов распределенной МАС), что порядок, в котором агенты совершают свои операции, всегда такой, что конфликты не возникают. В классических МАС описываются изменения глобального состояния, в то время как в распределенных МАС действия описывают изменения локального состояния.

МАС были успешно использованы при определении формальной семантики различных языков программирования и моделирования, а также непроцедурных языков спецификаций, среди которых объектно ориентированные языки программирования *Java* [52] и *C#* [85], язык описания вычислительных систем *SystemC* [136], язык *SDL* для спецификации и описания поведения реактивных и распределенных систем [138], универсальный язык моделирования *UML* [95, 109], язык описания аппаратуры *VHDL* [137] и т.д. Аппарат МАС можно рассматривать как формальную *рамочную* технологию определения самых разнообразных моделей вычислений, от конечных автоматов и до машин Тьюринга [84]. В работе [139] рассматривается моделирование высокоуровневых сетей Петри на основе МАС. Большая коллекция работ по тематике МАС представлена на сайте Мичиганского университета [74].

Привлекательным (хотя и нашедшим свое применение лишь в

специальных предметных областях) методом задания формальной семантики являются системы переходов состояний (СПС) [81]. Часто СПС расширяются определенной дополнительной функцией разметки состояний и называются в этом случае структурами (или моделями) Крипке [55]. Структуры Крипке играют важную роль в верификации на основе метода проверки моделей (*model checking*).

Структурой Крипке над множеством атомарных высказываний AP называется четверка [55]

$$M = (S, S_0, R, L) ,$$

где S – конечное непустое множество состояний; $S_0 \subseteq S$ – множество начальных состояний; $R \subseteq S \times S$ – тотальное отношение переходов на S. Это означает, что для каждого состояния $s \in S$ должен существовать хотя бы один потомок $s' \in S$, т.е. имеет место $R(s,s')$; L: $S \rightarrow 2^{AP}$ – функция, которая помечает каждое состояние множеством атомарных высказываний, истинных в этом состоянии.

Путь в структуре Крипке из состояния s_0 – это бесконечная последовательность состояний $\pi = s_0 s_1 s_2 \ldots$, такая, что для всех $i \geq 0$ выполняется $R(s_i, s_{i+1})$.

Описательные возможности СПС значительно возрастают при использовании: а) составных (векторных) состояний, когда каждый компонент состояния описывает подсостояние какого-то компонентного элемента [55]. Данный подход позволяет, например, легко представить в виде СПС сети Петри [252]; б) композиционных методов построения СПС. В этом случае система специфицируется как множество модулей. Предполагается, что разработаны структуры Крипке для каждого из них. Существует два основных пути конструирования общей структуры Крипке: 1) синхронная композиция и асинхронная (интерливинговая) композиция [55, 106]. При использовании синхронной композиции все компоненты системы изменяют свое состояние одновременно. В случае же асинхронной композиции в каждый момент времени только один компонент изменяет свое состояние. Таким образом, с использованием композиционных методов можно моделировать как синхронные, так и асинхронные системы.

СПС и структуры Крипке наиболее пригодны для описания семантики в тех случаях, когда в языке явно или неявно присутствует понятие состояния. Как правило, это языки спецификаций систем, аппаратуры, реактивных, управляющих и гибридных систем. Традиционные языки программирования ориентированы на описание

действий, поэтому для них данный метод не вполне приемлем.

Характерным языком, семантика которого наиболее адекватно может быть выражена с использованием СПС, является язык автоматных программ [66]. Удачное применение нашли СПС и для определения семантики некоторых диаграмм универсального языка моделирования *UML*. Так, например, в [126] представлена формальная семантика диаграмм активности *UML* с использованием синхронизируемых помеченных структур Крипке. В качестве другого примера можно привести язык *POOSL* (параллельный объектно-ориентированный язык спецификаций) [211], являющийся языком спецификаций системного уровня и предназначенный для моделирования и анализа сложных программно-аппаратных систем реального времени. Семантика *POOSL* определяется с помощью временной помеченной системы переходов. Эта семантика может формально описывать параллельность, распределенность, взаимодействия, время и функциональные свойства системы в единой выполнимой модели, используя небольшое множество очень мощных примитивов.

Большим преимуществом использования структур Крипке для спецификации формальной семантики языка являются развитые методы и имеющиеся в наличии средства для верификации и анализа программ на их основе (например, *SMV* [93, 186]). В работе [254] определен класс МАС, для которых возможна трансформация в структуры Крипке, а также определены собственно правила преобразования.

Таким образом, на основе изложенного выше анализа выбираем два взаимосвязанных подхода к определению формальной семантики ФБ. Первый подход основан на модифицированных распределенных МАС, а второй подход – на СПС. В то же время в силу рамочного характера МАС «семантическая» СПС (во втором подходе) может быть представлена в форме МАС.

1.4.3. Проблема анализа и верификации проектов IEC 61499

Верификация является одним из основных этапов в проектировании управляющего программного обеспечения (ПО) [10, 53, 63], что, в частности, объясняется повышенными требованиями к надежности ответственных систем управления. Под *верификацией* понимается проверка корректности и правильности логического поведения системы. Данный термин покрывает довольно широкую об-

ласть исследований в области ПО, включая экспертизу, статический анализ, тестирование, мониторинг, имитационное моделирование [60]. В дальнейшем будем касаться только *формальной* верификации, поскольку именно она (в отличие, например, от тестирования) обеспечивает полную проверку работоспособности программы [53].

Формальная верификация программ – это приемы и методы формального доказательства (или опровержения) того, что модель программной системы удовлетворяет заданной формальной спецификации [10, 53]. В настоящее время методы формальной верификации разрабатываются в трех основных направлениях [53]: 1) дедуктивная верификация [64]; 2) проверка эквивалентности; 3) проверка моделей (*model* checking) [63]. Не вдаваясь в подробности, отметим, что наибольший прогресс достигнут пока только в третьем направлении [53]. В связи с этим следует напомнить, что в 2008 г. Ассоциацией по вычислительной технике (ACM) трем ученым (Э. Кларку, А. Эмерсону и И. Сифакису) была вручена высшая награда Ассоциации – премия Тьюринга «За их роль в превращении *метода Model checking* в высокоэффективную технологию верификации, широко используемую в индустрии разработки программного обеспечения и аппаратных средств».

Проверка моделей (*Model checking*) – это метод проверки того, что на заданной формальной модели поведения системы заданное логическое свойство (требование) выполняется [10, 53, 63]. Это определение справедливо для любой логики и любого класса формальных моделей поведения. Впервые эта техника была разработана для моделей систем переходов и временных (темпоральных) логик. Обычно в этой технике верификации используется алгоритм проверки формул временных логик *LTL* и *CTL* [10]. Нового качественного уровня проверка моделей достигла при использовании упорядоченных двоичных решающих диаграммах (*OBDD*) для сжатого представления пространства состояний и символьных алгоритмов верификации моделей, основанных на манипуляциях с булевыми формулами [176]. Использование *OBDD* позволяет верифицировать некоторые системы, которые имеют более чем 10^{20} состояний [91]. Различные модификации методов, основанных на *OBDD*, могут расширить число состояний до 10^{120} и более [92].

Существует целый ряд инструментальных систем поддержки метода *Model checking*, в том числе системы S*PIN*, *SMV*, *YASM*, *Uppaal*, *Kronos*, *LTSA*, *ORIS*, *PRISM* и т.д. Достаточно полные обзоры инструментальных средств представлены в [53, 63]. Сравнение

верификаторов *SMV*, *SPIN*, *SPIN+PO*, *INCA* на основе экспериментов с обнаружением тупиков в моделях 17 параллельных программ приведено в работе [110]. Согласно результатам этой работы, эффективность верификаторов зависит от класса решаемых задач, но в целом можно сделать вывод, что верификатор *SMV* по времени решения задач намного более эффективный, чем *SPIN* и в большинстве случаев эффективнее *SPIN+PO* (т.е. верификатора *SPIN*, дополненного пакетом для поддержки метода частичного порядка).

Среди инструментальных средств поддержки *Model checking* можно выделить систему *SMV* [104, 180]. Основные преимущества этой системы: 1) использование технологий *OBDD* и *SAT* для проверки моделей, что ускоряет верификацию и позволяет исследовать модели большой размерности; 2) поддержка модульности и синхронно-асинхронной семантики выполнения, что позволяет моделировать широкий класс систем; 3) простой синтаксис. Одна из реализаций *SMV* (*NuSMV* [104]), кроме верификации, позволяет проводить также и имитационное моделирование. Следует отметить, что система *SMV* успешно использовалась для верификации во многих областях разработки аппаратуры и ПО [10, 53]. Два из многочисленных примеров использования системы *SMV* в предметной области, связанной с распределенными системами управления: 1) при верификации системы управления АЭС на основе ФБ стандарта IEC 61131-3 [256] и 2) при верификации диаграммы состояний языка *UML* [107].

Ниже приводится краткий обзор конкретных подходов к верификации ФБ IEC 61499, предложенных в литературе и используемых на практике. Как отмечалось выше, наибольшую популярность для моделирования ФБ получили *NCES*-сети [150], являющиеся расширением сетей Петри. Исторически они явились первым математическим аппаратом, используемым для формального описания ФБ. Для их анализа был построен верификатор *VEDA*, основанный на построении графа достижимости [249]. *NCES*-сети использовались для моделирования ФБ в рамках немецкого проекта *VAIAS* [245],
а также в европейском проекте *TORERO* [175]. В последней работе[175] анонсировалась новая инструментальная система *Function Block System Developer* (*FBSD*) для анализа ФБ на основе *NCES*-сетей, однако, вероятно, система не получила должного развития и информация о ней в настоящее время недоступна. В работах [137, 240] предложены методы анализа *NCES*-сетей на основе *Model checking*, на ос-

нове которых разработан верификатор *NCES*-сетей *SESA* [209]. В дальнейшем (2008 г.) для поддержки разработки *NCES*-моделей была разработана инструментальная система *ViVe* [234], включающая в свой состав данный верификатор. Система *ViVe* предлагает развитый графический пользовательский интерфейс.

В 2010 г. на основе *Model checking* на языке *SWI-Prolog* [220] разработана система анализа временных ингибиторных *NCES*-сетей *TNCES-Workbench* [229]. Данная система позволяет вычислять граф достижимости и анализировать его в дальнейшем с использованием формул временной логики, но практическое применение интерпретатора формул временной логики в случае графа достижимости большой размерности проблематично вследствие невысокой скорости выполнения Пролог-программ.

В работе [201] предлагается подход к моделированию ФБ на основе арифметических *NCES*-сетей. Для анализа сетей данного вида предлагается их преобразование в структуру Крипке (через промежуточную *A*-модель) [17] с последующим кодированием и использованием верификатора *SMV* [16]. В работе [113] предлагается комбинированное решение, включающее использование сетевых моделей *SIPN* и языка *SMV* для моделирования и верификации ФБ.

Моделирование и верификация ФБ с использованием временных автоматов представлены в работах [168, 215]. Причем в работе [215] в качестве верификатора использовалась система *Uppaal* [105]. В работе [96] предлагаются расширенные автоматы для моделирования ФБ с учетом их модели выполнения. Для верификации моделей ФБ (учитывающих дисциплину их выполнения) на основе расширенных автоматов [96] предлагается использование общецелевого средства для синтеза и верификации дискретно-событийных систем *Supremica* [76], в котором реализованы алгоритмы теории супервизорного управления [202].

Из полуформальных моделей для представления и верификации ФБ можно отметить языковые средства. Синхронные языки программирования использовались в работах [129] (язык *Signal*) и [259] (язык *Esterel*). В первой работе для верификации использовалось *CASE*-средство *SILDEX*, а во второй работе – средство *Design Verifier*, входящее в инструментальную систему *Esterel Studio*, причем для верификации применялся метод синхронных наблюдателей, позволяющий обходиться без использования временной логики. Как продолжение этой работы в [212] предлагается подход для формальной верификации проектов IEC 61499 с использованием на-

блюдателей, представленных в виде ФБ. В рамках этого подхода разработаны два алгоритма: один на основе проверки моделей на основе *CTL* с использованием таблиц, а второй – на основе построения графа достижимых состояний. В работе [122] для представления системы переходов ФБ, разработанной в [48], предлагается язык *SWI Prolog* [220]. На основе системы переходов ФБ строится граф достижимости, который исследуется с использованием интерпретатора *CTL*-формул.

В работе [87] начальное описание систем управления производится на основе объектно ориентированного подхода с использованием языка *UML*. В дальнейшем описание мехатронных объектов трансформируется в систему ФБ стандарта IEC 61131-3 или IEC 61499, которая затем преобразуется в программу на входном языке верификатора *SMV*. Однако эта работа остается на уровне идеи и никак не детализируется.

В ряде случаев для верификации специфичных свойств ФБ создаются специальные методы и средства. Например, в работе [79] для проверки робастности (устойчивости) ФБ относительно порядка приема входных сигналов на основе теории конечных автоматов разработаны соответствующая теория и алгоритмы. Данная активность была аргументирована непригодностью других методов верификации.

Следует отметить, что хороший обзор методов верификации ФБ IEC 61499 был дан в работе [151]. Однако данный обзор не является вполне полным, например, в нем не рассматриваются полуформальные методы на основе языков программирования. Помимо этого, краткий обзор работ по верификации ФБ можно найти в [26].

Из представленного выше обзора и анализа можно сделать следующие выводы:

1) формальная верификация ФБ тесно связана с методами их формального описания. Метод верификации ФБ, как правило, определяется выбранной моделью поведения и инструментальной системой, которая поддерживает данную поведенческую модель;

2) в большинстве работ речь не идет о каких-то новых методах верификации, в них скорее адаптируются существующие методы, но в отношении ФБ, с учетом их особенностей;

3) можно выделить три «технологических» подхода к верификации систем ФБ:

− с использованием промежуточной модели ФБ и ее непосредственным исследованием;

− с использованием промежуточной модели ФБ и ее дальней-

шим преобразованием;

− без использования промежуточной модели.

В первом случае описание ФБ преобразуется в некоторую промежуточную формальную модель (например, *NCES*-сеть), которая в дальнейшем исследуется специальным верификатором, ориентированным на эту модель. Во втором случае промежуточная формальная модель (например, *aNCES*-сеть) дополнительно преобразуется в программу на входном языке выбранного верификатора. Третий вариант предполагает прямое преобразование описания ФБ в программу на входном языке верификатора. Как правило, этот входной язык является формальным (имеет формальную семантику) и его в определенной мере можно считать целевой формальной моделью.

1.4.4. Описание и анализ систем с использованием онтологий

Система управления на основе стандарта IEC 61499 может представлять собой сложную многоуровневую иерархическую систему с множеством связей. Проверить такую запутанную структуру вручную очень непросто и порой даже невозможно. Важным этапом в проектировании систем управления является верификация, призванная проверить свойства корректности системы. Но верификация сложна и на практике зачастую этот процесс игнорируется и подменяется традиционным тестированием.

Все большую роль в выявлении ошибок на стадиях проектирования, предшествующих внедрению, приобретает *семантический анализ* проектов, предполагающий проверку определенных семантических ограничений, отражающих глубинные отношения между артефактами проектирования, не поддающиеся синтаксическому анализу. Семантический анализ не требует интенсивных и ресурсоемких вычислений, какие необходимы при верификации, но при этом способен обнаруживать серьезные ошибки, которые не являются очевидными. В дальнейшем будем касаться только той области семантического анализа, которая относится к языкам, включая языки программирования. Синтаксическими свойствами языка являются те свойства, которые описываются грамматикой, а семантическими свойствами – те, которые грамматикой не описываются [61]. Семантические свойства обычно описываются на естественном языке. Однако отсутствие формализации семантических свойств может порождать различные проблемы. Семантический анализ является одной из фаз работы (традиционного) компилятора. Данная фаза следует за фазой синтаксического анализа и предшест-

вует фазе генерации кода [61]. На фазе семантического анализа используются семантические свойства языка.

В проектировании ПО большое распространение нашли статические методы анализа, основанные, прежде всего, на *графах зависимостей* [54]. Граф зависимостей служит мощным инструментом для понимания программ, их поддержки, отладки, тестирования, поиска аномалий и ошибок, оптимизации, а также распараллеливания. Основными источниками зависимостей являются управление и данные. Методы анализа зависимостей начали свое развитие с 70-х гг. XX в. и успешно развиваются до настоящего времени. Существует множество работ по данному направлению (например, [11, 77, 199, 214]). Большинство работ ориентировано на уровень реализации для традиционного ПО. Например, в работе [199] определяется формальная модель зависимостей программ, обобщаются зависимости по данным и управлению, вводится понятие синтаксической и семантической зависимостей. В то же время в работе [214] предложен подход к анализу зависимостей ПО на архитектурном уровне, относящимся к более высокому уровню абстракции. Данный подход проиллюстрирован на примере ПО газораспределительной станции, где были найдены аномалии и локальные ошибки. В работе [77] для анализа встроенных систем реального времени предлагаются графы зависимостей событий, позволяющие оценить временные характеристики системы. В работе [11] графы зависимостей рассматриваются как основа для распараллеливания программ. Построение и анализ графов зависимостей по управлению и по данным для систем ФБ могут быть проведены в рамках семантического анализа.

В настоящее время в связи с появлением и внедрением в практику мультиагентных систем и развитием концепции семантического *Web* большое значение приобретает онтологическое представление знаний [12]. Под *онтологией* понимается формальное представление множества концептов внутри домена и отношений между этими концептами [12]. Наиболее популярным формализмом *Web*-онтологий в настоящее время является дескриптивная логика (ДЛ) [224]. ДЛ позволяет описывать как структуры и ограничения, так и осуществлять логический вывод, с помощью которого определяется как синтаксическая, так и семантическая корректность описания системы. ДЛ является формализмом для представления знаний, важнейшим свойством которого является разрешимость. Системы ДЛ имеют семантику «открытого мира», что позволяет специ-

фицировать неполные знания. В зависимости от набора конструкторов различают несколько видов ДЛ. Система представления знаний на основе ДЛ предлагает ряд типовых рассуждений о терминологии и утверждениях. К задачам вывода относятся выполнимость концепта, категоризация, проверка экземпляров и отношений. База знаний системы представления знаний (СПЗ) на основе ДЛ состоит из двух компонентов: *TBox* и *ABox*. Компонент *TBox* вводит терминологию, иными словами, словарь предметной области, в то время как компонент *ABox* содержит утверждения об именованных представителях (экземплярах) концептов в терминах словаря. Составными частями *TBox* являются концепты и роли.

На основе ДЛ был разработан язык *Web*-онтологий *OWL* [250], являющийся одной из основных составных частей семантического *Web*. Язык *OWL* имеет три диалекта. С использованием диалекта *OWL DL* может быть представлена любая формула ДЛ типа *SHOIN*. В настоящее время в рамках развития языка *OWL* предложен язык *OWL* 1.1, в основу которого положены ДЛ типа *SROIQ* [159], а также язык *OWL* 2 [143]. Существует несколько реализаций систем ДЛ. Как правило, такие системы включают клиентскую часть для поддержки проектирования и визуализации *Web*-онтологий и серверную часть, реализующую механизм рассуждений (*reasoner*). Наиболее популярной клиентской частью в настоящее время является система *Protégé* [201].

Для увеличения выразительной силы в совокупности с ДЛ используется логика хорновских дизъюнктов, являющаяся фрагментом логики первого порядка [62]. Хорновские дизъюнкты могут представляться в виде правил языков логического программирования, например, *Prolog* или *Datalog*. В качестве языка правил в семантическом *Web* используется язык *SWRL* [221]. Для того, чтобы формализм *OWL DL + SWRL* был разрешим, следует использовать так называемые «ДЛ-безопасные» правила (*DL-safe rule* [184]). Безопасные правила ограничены до известных индивидуальностей (экземпляров).

Ниже приведен краткий обзор работ по использованию онтологий в проектировании ФБ и в программной инженерии в целом. В работе [194] проиллюстрирована идея по «обогащению» эталонной модели ФБ с использованием онтологий. Однако основной целью данной работы было использование семантического описания ФБ для автоматического поиска и обнаружения ФБ в приложениях, основанных на *Web*-сервисах. Поэтому представленная модель он-

тологии ФБ не детальна и недостаточна для проведения семантического анализа. Возможность использования семантического *Web* и сервис-ориентированных архитектур (*SOA*) в промышленной автоматике обсуждается в [167]. Эта работа вызвана проблемами интероперабельности, масштабируемости, поддержки технологии *plug-and-play* и легкостью интеграции. В статье сделан вывод, что использование *Web*-сервисов увеличит интеллектуальность систем автоматизации. В работе [141] описывается подход к генерации кода для IEC 61499 на основе итерационной базы знаний, представленной в форме *XML*, и расширенных форм Бэкуса–Наура (РФБН). Подход к генерации проектов на основе ФБ с использованием технологий семантического *Web* предложен в работе [205]. Данный подход основан на использовании некоторых специальных шаблонов проектирования. Однако в данном случае используются ФБ, не относящиеся к IEC 61499, а, собственно, онтология ФБ не представлена. Можно попытаться провести аналогию онтологического описания ФБ с онтологическими описаниями других абстрактных моделей. Например, в [233] предлагается многоуровневая онтология статики и динамики высокоуровневых сетей Петри. Данная онтология может использоваться для синтаксического и семантического анализа сетевых моделей. Среди немногочисленных работ по семантическому анализу можно выделить работу [80], в которой предлагается метод анализа текстов на естественном языке с использованием онтологий.

Существует несколько исследовательских групп, занимающихся внедрением онтологий в процесс разработки доменно-специфических языков (ДСЯ) [89, 145, 251]. Язык ФБ, без сомнения, относится к ДСЯ, но поскольку данный язык существует де-факто, данное направление исследований в целом особого интереса по обозначенной выше теме не представляет. Хотя предложенный метод интеграции ДСЯ на основе онтологий [89] представляет несомненный интерес для дальнейших исследований по использованию онтологии ФБ в проектировании управляющего ПО для разных предметных областей.

Следует отметить все большее распространение и использование онтологий в проектировании «традиционного» ПО [135, 152, 193, 255]. В работе [255] выделены следующие области использования онтологий в этой сфере: 1) разработка требований; 2) повторное использование компонентов; 3) интеграция с языками моделирования ПО (имеется ввиду языки *MOF/UML* [192]); 4) разработка до-

менной объектной модели; 5) поддержка кодирования; 6) документирование кода; 7) разработка ПО среднего уровня; 8) представление бизнес-логики; 9) разработка семантических *Web*-сервисов; 10) поддержка проектов; 11) автоматическое обновление ПО; 12) тестирование и отладка. Проектирование управляющего ПО на основе ФБ IEC 61499 имеет существенные отличия от проектирования традиционного ПО на основе объектно ориентированного подхода, поэтому перечисленные выше области применения онтологий, хотя в общем, остаются действительными, но требуют переосмысления.

Использование *Web*-онтологий в проектировании (и в семантическом анализе, в частности) имеет следующие *преимущества*: 1) формальный аппарат *Web*-онтологий позволяет точно, ясно и на высоком уровне выразить семантические свойства; 2) формальное представление семантических свойств позволяет использовать формальный аппарат логического вывода для доказательства этих свойств, что в свою очередь повышает достоверность полученных результатов анализа; 3) использование дескриптивной логики и *DL*-безопасных *SWRL* правил гарантирует разрешимость задачи классификации (в нашем случае – анализа), т.е. задача анализа должна закончиться за приемлемое время; 4) дескриптивная логика имеет семантику «открытого мира», что позволяет описывать неполные знания, которые могут быть дополнены в последующем. Добавление новых экземпляров не может изменить классификацию для старых экземпляров; 5) системы семантического анализа на основе онтологий проще перепроектировать, поскольку этот процесс сводится к изменению онтологии, а не к изменению программы; 6) *Web*-онтология является разделяемым ресурсом и может быть размещена в сетях Интернет/Интранет для использования людьми или программами.

Все вышесказанное позволяет констатировать, что использование технологий семантического *Web* и, в частности, онтологий является весьма перспективным направлением в проектировании систем управления на основе IEC 61499.

1.5. Методы проектирования систем управления на основе IEC 61499

Одной из причин, тормозящих внедрение в промышленную практику распределенных систем управления на основе IEC 61499, является отсутствие методологии проектирования систем данного класса. Многие понятия стандарта оказались незнакомы инженерам

по управлению, например, идея распределенного приложения, управления выполнением на основе потока событий, СИФБ [241, 263]. Для преодоления этого «образовательного» провала на первых порах было предложено несколько *шаблонов проектирования* систем управления на основе ФБ, среди которых «Модель/Представление/Контроллер» (*MVC*), «Распределенное приложение», «Прокси» [101].

Обычно выделяют следующие основные фазы создания ПО: 1) формирование требований к системе; 2) проектирование; 3) реализация; 4) тестирование; 5) ввод в действие; 6) эксплуатация и сопровождение [3]. Как уже было отмечено выше, проекты на основе IEC 61499 имеют ряд отличий от традиционных программных проектов. С учетом этого процесс проектирования распределенных систем управления на основе IEC 61499 в работе [241] представлен следующей последовательностью этапов: 1) определение требований; 2) спецификация процессов; 3) выбор аппаратной части; 4) разработка архитектуры ПО; 5) проектирование алгоритмов и протоколов; 6) реализация: кодирование и отладка; 7) верификация и валидация; 8) спецификация коммуникационной сети и устройств; 9) размещение; 10) сопровождение и реконфигурация. Как отмечается в этой работе, IEC 61499 хорошо поддерживает те этапы разработки, которые находятся в последней части представленного «спектра». Следовательно, необходимо найти дополняющие методы и средства проектирования, поддерживающие проектирование в начальной части спектра.

Интенсивные исследования в области разработки методов проектирования в области систем управления на основе IEC 61499 начались с середины 2000 г. Много идей было заимствовано из области проектирования ПО. Доминирующей методологией разработки ПО на сегодняшний день является объектно ориентированный анализ и проектирование [6], основным языком данного подхода является *UML* [7]. Язык *UML* может быть использован практически на всех этапах разработки – от определения требований и анализа и до кодогенерации и размещения кода [6, 7]. Основной тенденцией в проектировании ПО является проектирование на основе управления моделями. Состояние дел в этой области приводится ниже.

1.5.1. Объектно ориентированное проектирование

Исследования в области проектирования систем управления промышленными системами на основе объектно ориентированного подхода начались в 90-е гг. К числу этих работ относятся также ис-

следования, связанные с использованием языка *UML* [14]. Наиболее известным расширением *UML* для разработки сложных систем реального времени является *UML-RT* [207]. В качестве другого расширения *UML* для моделирования систем управления можно привести язык *CSML* [260]. Ряд исследователей пошел по пути совместного использования *UML* с альтернативными средствами описания систем, например, диаграммами потоков данных [86], диаграммами *SDL* [171] и т.д. Недостатками использования языка *UML* в разработке систем управления промышленными процессами являются: 1) слабые знания инженеров по системам управления в области объектно-ориентированного проектирования и других передовых программных технологий; 2) определенные сложности реализации потоков событий; 3) трудности генерации полноценного управляющего приложения.

В последние годы предпринимаются попытки сближения парадигмы проектирования на основе ФБ и *UML* [156, 196, 230, 261]. В работе [261] приводится подробное сравнение этих двух подходов и делается вывод, что они в определенных аспектах очень близки между собой и первая парадигма является подмножеством второй. В работе [230] предлагаются преобразования определенных *UML*-диаграмм (диаграмм вариантов использования и диаграмм последовательностей) в сети ФБ. На основе данных преобразований разработана система поддержки проектирования распределенных управляющих систем *CORFU*. В работе [156] предлагается профайл для адаптеров ФБ, который отвечает за соединение портов языка *UML-RT* и языков, ориентированных на ФБ IEC 61131-3, IEC 61499 или *Matlab/Simulink*. Работа [196] демонстрирует использование *UML* для решения множества практических задач, относящихся к проектированию систем автоматизации на основе IEC 61499. Консорциумом OMG на основе языка *UML* разработан предметно-ориентированный язык моделирования систем *SysML*, ориентированный на использование в инженерных приложениях [191]. Работа [157] посвящена разработке методов связывания технологии ФБ с *SysML*. Показательно, что инициатива *IMS OOONEIDA* [244] нацелена на интеграцию усилий разработчиков промышленных систем управления на основе открытых стандартов IEC 61499, *UML*, IEC 61131-3.

Следует отметить, что, несмотря на активизацию деятельности по сближению отмеченных выше подходов, работы в этом направлении далеки от завершения, в частности, не до конца разработаны методы и средства трансформации представлений на основе *UML*

и ФБ стандарта IEC 61499. Как показала практика, между парадигмами проектирования на основе *UML* и ФБ существует разрыв, затрудняющий реализацию систем управления на основе ФБ IEC 61499 при использовании их *UML*-спецификаций. Одной из его причин являются разные подходы при описании программных систем и бизнес-процессов (язык *UML*) и структурных схем (язык ФБ). Это требует новых исследований для сближения языка *UML* с языком ФБ.

1.5.2. Подход на основе управления моделями

В проектировании ПО большое распространение нашел подход на основе управления моделями (или иначе – на основе *потока моделей*) [182]. Отправным пунктом проектирования является начальная модель системы, а конечным результатом – целевая модель системы. Технология, основанная на управлении моделями (*Model Driven Engineering – MDE*), является в настоящее время передовой технологией разработки программного обеспечения. Трансформация моделей является «душой и сердцем» этой технологии [208]. Группа *OMG* [190] предложила архитектуру потока моделей (*Model Driven Architecture – MDA*) для интеграции различных средств *MDE*. Для определения моделей и метамоделей консорциум *OMG* установил хорошо известные стандарты *MOF* и *UML*. Для стандартизации трансформации моделей *OMG* анонсировала проект стандарта *RFP MOF* 2.0 *Query/Views/Transformation*, который включает требования к языкам трансформаций. Архитектура *MDA* основана на идее автоматической трансформации абстрактных, платформо-независимых моделей (*PIM*-моделей) в платформо-специфические модели (*PSM*-модели). Эти трансформации специфицируются множеством трансформационных правил, которые могут быть повторно использованы для множества похожих программных проектов. Технология *MDE* может быть перенесена из традиционной области проектирования ПО в другие области проектирования. Например, современные вызовы в области автоматизации в случае использования методологии разработки, управляемой моделями, обсуждаются в [217].

Для того, чтобы трансформировать модели, они должны быть выражены на некотором языке моделирования, синтаксис которого определяется некоторой *метамоделью*. Метамодели выражают как исходную, так и целевую модели трансформации. Различают эндогенную и экзогенную трансформации. *Эндогенные трансформации* – это трансформации между моделями, выраженными в одной и той же метамодели. *Экзогенные трансформации* – это трансформа-

ции между моделями, выраженными в разных метамоделях. Различают также горизонтальную и вертикальную трансформации. При *горизонтальной трансформации* исходная и целевая модели находятся на одном и том же уровне *абстракции*, *а при вертикальной трансформации* – на разных. Примером эндогенной горизонтальной трансформации является рефакторинг. Многие задачи синтеза можно отнести к экзогенной вертикальной трансформации.

В широком смысле под *рефакторингом* программного обеспечения (ПО) понимается изменение его внутренней структуры без изменения внешнего поведения с целью повышения качества ПО [133]. Рефакторинг определяется на различных уровнях представления ПО – от представления в виде кода и кончая представлением в виде моделей. Рефакторинг является одним из элементов поддержки эволюции программных систем. Данному виду трансформации ПО посвящен ряд публикаций, из которых следует выделить работу, касающуюся рефакторинга на основе преобразований графов [179].

В работе [173] предлагается подход, который расширяет *MDA* в области доменно-специфических языков моделирования. Данный подход получил название интегрированных модельных вычислений (*Model-Integrated Computing – MIC*). Мехатронные системы сложнее программных систем в том плане, что они содержат также электронные и механические компоненты. Поэтому их разработка имеет специфические особенности, что предполагает широкое использование доменно-специфических моделей. Одной из первых работ, в которой сделана попытка перенести *MIC*-подход на область мехатронных систем, является работа [227]. В ней предлагается архитектура, которая предоставляет не только артефакты этапа реализации, но и этапов анализа и проектирования в процессе разработки. Однако в данной работе идея трансформации моделей явно не раскрывается.

К технологии *MDE* в определенной мере можно отнести и использование *шаблонов проектирования*. Шаблоны проектирования определяют наиболее важные и повторяющиеся приемы программирования, что способствует увеличению степени повторности использования в проектировании и программировании. Шаблонам проектирования ПО посвящен ряд книг и публикаций, наиболее часто шаблоны проектирования используются в объекто ориентированном проектировании [13]. Следует отметить, что ранее предлагались шаблоны проектирования управляющего ПО на основе стандарта IEC 61499 [101]. В работах [39, 40, 116] обсуждаются шаблоны проектирования IEC 61499 «производитель–потребитель»,

«работа с распределенной базой данных», «распределенная реализация сетей Петри». В работе [90] предложена концепция интеллектуальных *объектов автоматизации* для представления мехатронных частей систем. Эти объекты автоматизации реализованы с использованием шаблона проектирования *MVC* и включают в себя как управляющий компонент, так компоненты, осуществляющие визуализацию и имитационное моделирование.

Подход на основе управления моделями еще не нашел должного развития в сфере проектирования на основе IEC 61499, но он уже начинает набирать обороты [228]. В качестве примера также можно сослаться на европейский проект *MEDEIA* (*Model-Driven Embedded System Design Environment for the Industrial Automation Sector*) [178]. Однако в данном проекте не затрагиваются все аспекты проектирования систем управления промышленной автоматики на основе управления моделями. В частности, не рассматриваются вопросы рефакторинга и синтеза формальных моделей систем управления, не формализуются правила трансформации моделей.

Перспективным подходом к трансформации моделей является подход, основанный на трансформации графов [123]. Данный подход развивается и уже нашел применение в *MDE* [144]. Существуют хорошие предпосылки для использования *MDE*, *MIC* и графовых преобразований в проектировании распределенных компонентно-базированных систем управления промышленными процессами на основе стандарта IEC 61499. В виде графов могут быть представлены как составные ФБ, приложения и субприложения, так и базисные ФБ, основой которых является диаграмма управления выполнением (диаграмма *ECC*). Преимуществом подхода является его формальность, что позволяет избежать многих ошибок проектирования и проверить правила на корректность. Этот подход имеет смысл взять в данной работе за основу. В качестве метамоделей при этом используются типизированные атрибутные графы (ТАГ) [123].

Дальнейшим развитием технологии *MDE* является технология на основе управления онтологиями (*Ontology Driven Engineering – ODE*) [200]. Это очень перспективная технология, и можно предположить, что основные усилия разработчиков распределенных систем управления на основе IEC 61499 в ближайшем будущем сосредоточатся именно на этом направлении.

2. UML-FB – визуальный язык для моделирования систем управления промышленными процессами на основе стандарта IEC 61499

В данном разделе сделана попытка для сокращения семантического разрыва между структурным и объектно ориентированным подходами к проектированию систем управления. Для достижения этой цели язык ООАП *UML* комбинируется со структурным языком ФБ.

2.1. Краткое описание языка UML-FB

Основные особенности

Особенностью языка *UML-FB* (или иначе – профиля *UML-FB*) является его близость стандарту IEC 61499. Все конструкции, представимые с использованием языка спецификации ФБ [163], являются представимыми в *UML-FB*. Язык *UML-FB* можно рассматривать, с одной стороны, как полноценный язык моделирования систем управления, с другой – как некоторое расширение языка спецификации ФБ стандарта IEC 61499 средствами *UML*. При использовании *UML-FB*, например, появляется возможность явного описания иерархии системы в виде диаграммы классов. Для описания функционирования системы могут использоваться диаграммы последовательностей и кооперации, отсутствующие в стандарте IEC 61499. Кроме того, появляется возможность использования диаграмм *UML* в качестве языка запросов (для валидации и тестирования модели) и языка структурных и функциональных ограничений (для использования в процессе разработки корректных моделей). Однако довольно жесткие правила использования диаграмм, составляющих профиль *UML-FB*, и его особая семантика определяют ограниченность *UML-FB* в плане применимости в других предметных областях.

Язык *UML-FB* является расширением языка *UML* в двух направлениях: 1) в «стандартном» направлении. Это выражается в том, что используется «легкое» расширение метамодели языка *UML* с помощью механизма стереотипов; 2) в «нестандартном» направлении. Это проявляется в небольшом изменении базовой семантики *UML*, например, во введении дополнительного отношения явного «структурного наследования», имеющего семантику, несколько от-

личную от семантики обычного отношения наследования. В этом случае формально наследуются не только атрибуты и методы родительского класса, но и все агрегированные в родительский класс классы и все отношения между ними. Следует, однако, отметить, что с «представительской» (внешней) точки зрения это не нарушает соответствия языка *UML-FB* метамодели *UML* (расширенной стереотипами). Семантика «структурного наследования» может быть просто реализована трансформационными методами изменения исходной *UML-FB*-модели.

Типы использованных диаграмм

В языке *UML-FB* используются следующие виды диаграмм: диаграмма классов, диаграмма последовательностей, диаграмма кооперации и диаграмма состояний.

Диаграмма классов используется для представления: 1) полной системной иерархии компонентов (по отношениям агрегации и наследования); 2) типов ФБ, субприложений, адаптерных интерфейсов, ресурсов и устройств; 3) конфигураций систем и устройств; 4) интерфейсов ФБ, включающих событийные и информационные входы и выходы, а также *WITH*-связи между событийными и информационными линиями; 5) связей между ФБ, характерных для всех экземпляров заданных типов; 6) констант, представляющих параметры ФБ, субприложений, ресурсов и устройств.

С использованием *диаграмм последовательности* и *диаграмм кооперации* задаются связи между ФБ; между параметрами и ФБ, ресурсами и устройствами, а также представляются временные диаграммы последовательностей, определенные в ISO TR 8509, используемые для описания поведения сервисных интерфейсных функциональных блоков (СИФБ).

Диаграмма состояний используется для определения диаграммы управления выполнением (*ECC – Execution Control Chart*) базисного ФБ. При этом применяются простые диаграммы состояний, без использования составных и исторических состояний, а также сложных переходов.

Стереотипы классов

В основу стереотипизации классов *UML-FB* положены основные понятия стандарта IEC 61499. В диаграмме классов используются следующие стереотипы классов: *BFB* (базисный функциональный блок), *CFB* (составной функциональный блок), *SIFB* (сервисный интерфейсный функциональный блок), *SUB* (субприложение), *ADAPTER* (адаптерный интерфейс), *CONSTANT* (константа), *SER-*

VICE (сервис СИФБ), *SYSTEM* (система), *DEVICE* (устройство), *DEVICE_TYPE* (тип устройства), *RESOURCE* (ресурс), *RESOURCE_TYPE* (тип ресурса), *APPLICATION* (приложение), а также предопределенный стереотип *Interface*.

Стереотипы методов и атрибутов класса

Атрибуты классов *UML-FB* используются для представления информационных входов и выходов ФБ, субприложений, адаптеров и других артефактов стандарта IEC 61499, в то время как методы (операции) классов *UML-FB* – для определения событийных входов и выходов этих же объектов, а также *WITH*-ассоциаций.

Методы классов ФБ и субприложения типизируются с использованием стереотипов *INPUT* (входной) и *OUTPUT* (выходной). Атрибуты данных классов в дополнение к перечисленным могут иметь стереотипы *INTERNAL* (внутренний), *SOCKET* (адаптер-сокет) и *PLUG* (адаптер-штекер). Каждому атрибуту класса со стереотипом *INPUT*, *OUTPUT* или *INTERNAL* должен быть назначен тип (*Integer*, *Boolean*, *String* и т.д.), который будет использоваться при определении типа соответствующей переменной в ФБ. Атрибуты могут иметь начальные значения.

Сигнатура методов класса

Сигнатура методов класса используется для определения *WITH*-ассоциаций событийных и информационных входов и выходов. Название метода определяет имя событийного входа (выхода), а параметры (иначе, аргументы) метода – имена информационных входов (выходов), связанных с данным событийным входом (выходом) с использованием *WITH*-связей. Например, сигнатура метода *ei*1(*di*1, *di*2, *di*3) свидетельствует, что событийный вход *ei*1 связан *WITH*-связями с информационными входами *di*1,*di*2 и *di*3.

Отношения диаграммы классов

Используемые отношения в диаграмме классов – это отношения агрегации, наследования и ассоциации.

С помощью *отношения агрегации* полностью задается системная иерархия ФБ. При этом определяется не только иерархия классов (иначе – типов ФБ), но в сжатом виде задается и иерархия объектов (иначе – экземпляров ФБ). Размещение на ресурсе ФБ, входящих в приложение, производится с помощью агрегирующих связей, помеченных стереотипом *ALLOC*. Для включения в ФБ адаптера-штекера (адаптера-сокета) используется стереотип *PLUG* (*SOCKET*) агрегирующей связи. Для агрегативных связей задаются роли, имеющие важное значение в задании системы экземпляров ФБ (см. ниже).

Отношение ассоциации используется для определения событийных, информационных и адаптерных связей между ФБ на уровне их типов. Ассоциативная связь между классами в действительности определяет связь «каждый с каждым» между соответствующими экземплярами ФБ. Для ассоциативных связей применяются стереотипы *EVENT*, *DATA* и *ADAPTER*. Имена ролей ассоциативной связи могут использоваться для указания входов-выходов соответствующих экземпляров ФБ, соединенных событийной, информационной или адаптерной связью (что определяется стереотипом связи).

С помощью *отношения наследования* в системе ФБ может быть реализовано наследование событийных, информационных и адаптерных входов-выходов ФБ, а также интерфейсов в целом. Отношение наследования имеет несколько иной смысл между классами, представляющими ресурс (стереотип *RESOURCE*) и тип ресурса (стереотип *RESOURCE_TYPE*). В этом случае наследуются не только атрибуты и методы соответствующих родительских классов, но также внутренняя структура родительского класса, включая ФБ и связи между ними. То же самое относится и к классам, представляющим устройства. Дуга данного модифицированного отношения «структурного наследования» помечается стереотипом *STR*.

Представление объектов в диаграмме классов

Диаграмма классов в *UML-FB* может определять не только классы, но и объекты (экземпляры классов). При этом в *UML-FB* существует два различных случая: 1) представление экземпляров ФБ и 2) представление экземпляров устройств и ресурсов. Этим двум случаям соответствуют два разных метода представления: 1) с использованием связей; 2) с использованием специальных классов.

Экземпляры (типов) ФБ определяются в диаграмме классов с использованием агрегирующих связей. При этом роль агрегирующей связи определяет имя экземпляра ФБ. Число экземпляров ФБ, входящих в некоторый тип ФБ, определяется числом соответствующих агрегирующих связей. Имена экземпляров ФБ, которые входят в некоторый тип, должны быть попарно различны. На рис. 2.1 приведен показательный пример на эту тему. Как видно из этого рисунка, в тип *A* включены экземпляры *F*1, *F*2 и *F*3 типов *B*, *C* и *D* соответственно.

Экземпляры устройств и ресурсов (точнее, их конфигураций) представляются с использованием самостоятельных классов, связанных с соответствующим классом-типом с помощью связи наследования, помеченной стереотипом *STR*. Иллюстративный пример

приведен на рис. 2.2. На данном рисунке класс *T* является классом, определяющим тип устройства (ресурса), а класс *A* – классом, представляющим экземпляр устройства (ресурса) типа *T*. Наследование в данном случае подразумевает, что дочерний класс *A* добавляет к своему содержимому не только интерфейс родительского класса *T*, но также и его структурную «начинку», а именно: экземпляры ФБ *F*2 и *F*3, а также связи между ними. Результирующая конфигурация дочернего класса *A* структурно эквивалентна с диаграммой классов на рис. 2.1.

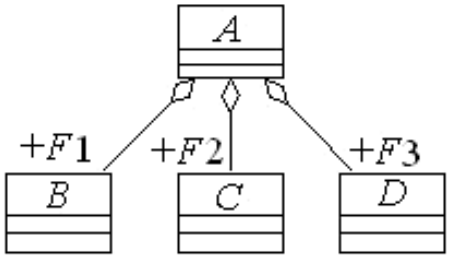


Рис. 2.1. Определение экземпляров ФБ с использованием агрегирующих связей

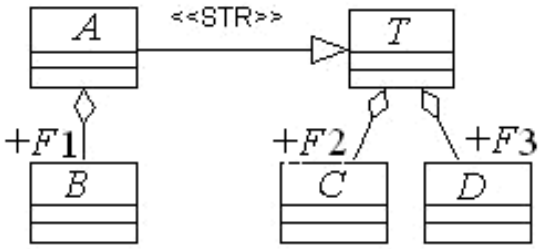


Рис. 2.2. Структурное наследование, связывающее типы и экземпляры устройств и ресурсов

Идентификация функциональных блоков

Экземпляры некоторого типа ФБ, включенные различные другие типы ФБ могут иметь в модели одинаковые имена (рис. 2.3). Поэтому существует необходимость выбора и применения некоторого недвусмысленного механизма идентификации ФБ. В общем случае (ссылаясь как пример на диаграммы на рис. 2.1 и 2.3) можно утверждать, что диаграмма классов в *UML-FB* представляет ациклический граф, в котором путь однозначно определяет объект. Для идентификации ФБ предлагается использование точечной нотации. При этом имя объекта, определенное с помощью точечной нотации, называется составным. Полное составное имя в действительности определяет путь от корня иерархии к требуемому объекту.

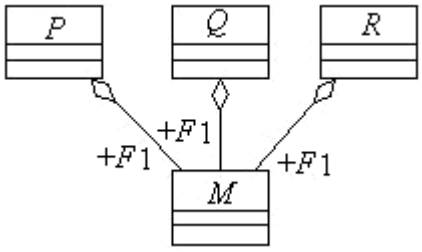


Рис. 2.3. Одноименные экземпляры *F*1 одного и того же типа *M*

Для упрощения именования возможно использование сокращенных имен. Сокращенное имя также определяет путь в графе. Требованием к сокращенному имени является однозначность идентификации объекта. Для его выполнения необходимо, чтобы первая компонента имени

однозначно идентифицировала некоторый объект в системной иерархии. Сокращенные имена могут быть расширены до полных имен. Назовем такую операцию нормализацией имен.

На приведенной на рис. 2.4 *UML-FB*-диаграмме класс *G* определяет два следующих объекта *F*1.*F*3.*F*4.*F*5:*G* и *F*1.*F*5:*G*. При этом первому объекту может быть присвоено сокращенное имя *F*4.*F*5:*G*, так как *F*4 является уникальным именем объекта на диаграмме классов. Класс *H* представляет объекты *F*2.*F7*:*H* и *F*1.*F*6.*F*7:*H*.

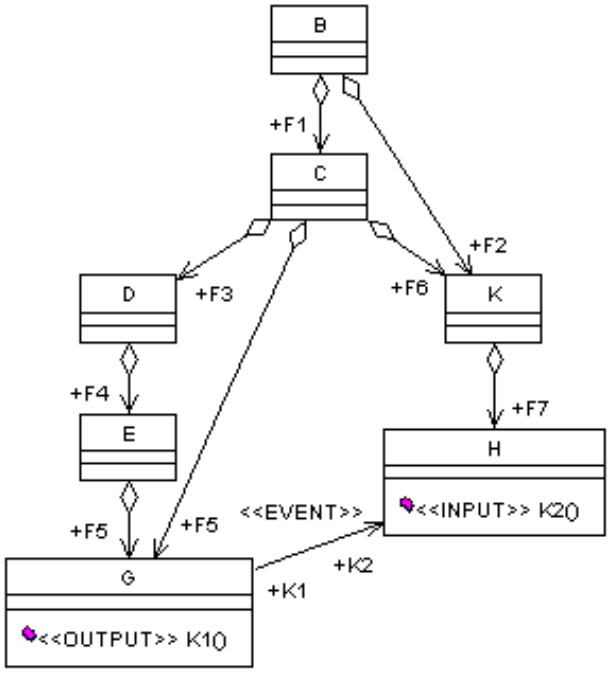


Рис. 2.4. Пример *UML-FB*-диаграммы классов

Множественные связи между классами

Ассоциативная связь между классами в диаграмме классов *UML-FB* в действительности определяет связь «каждый с каждым» между соответствующими объектами. Например, связь между классами *G* и *H* на рис. 2.4 определяет четыре связи между соответствующими объектами (рис. 2.5).

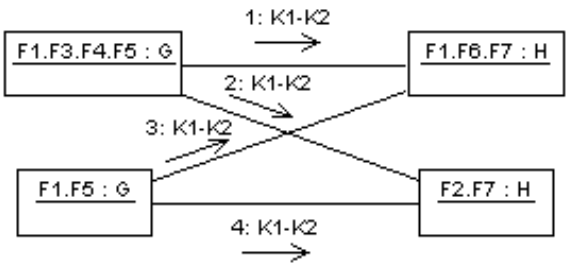


Рис. 2.5. Связь между классами *G* и *H* (см. рис. 2.4), раскрытая до уровня объектов

Именование сообщений в диаграммах последовательности и кооперации

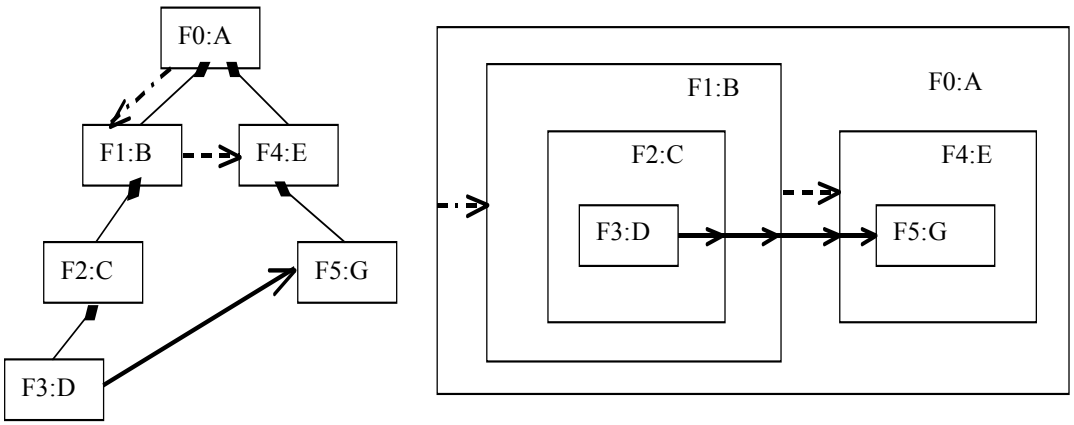
Имена сообщений в диаграммах последовательности и кооперации используются для уточнения задания связей между ФБ на уровне их входов-выходов. Имя сообщения имеет два формата: 1) *<имя выхода компонентного ФБ> – <имя входа компонентного ФБ>*; 2) *<имя входа и выхода компонентного ФБ>*. Первый вариант используется, если имена входов и выходов различаются, а второй – когда совпадают. Для диаграммы последовательности во втором случае может использоваться нотация вызова метода. ФБ-источник и ФБ-приемник определяются соответствующими объектами, которые связаны стрелкой сообщения. Например, на рис. 2.5 сообщение 4 с именем «K1-K2» определяет связь между выходом K1 компонентного ФБ *F*1*:F*5 типа *G* и входом K2 компонентого ФБ *F*2.*F*7 типа *H*.

Виды связей по локализации

В используемых диаграммах классов, последовательности и кооперации допускается использование различных видов (по критерию локализации) связей. Для иллюстрации данных видов связей используется дерево иерархии объектов, представленное на рис. 2.6,*а*. На рис. 2.6,*б* схематично приведен пример системы ФБ, сгенерированной по данному дереву.

Различаются локальные и транзитные связи. К *локальным связям* относятся прямые одноуровневые связи (связи между объектами-братьями) и прямые межуровневые связи. К первому классу относится, например, связь между объектом *F*1 и объектом *F*4, а ко второму – связь между объектами *F*0 и *F*1.

Под *транзитной связью* понимается связь между объектами, лежащими на разных уровнях и, возможно, на разных ветвях дерева иерархии. Стандарт IEC 61499 не позволяет использовать такого рода связи. Для реализации транзитной связи используется несколько межуровневых связей. Кроме того, транзитная связь предопределяет переопределение (дополнение) интерфейсов классов, через экземпляры которых она проходит транзитом. Например, для реализации транзитной связи (на рис. 2.6,*а* отмечена жирной линией) между объектами *F*3 и *F*5 требуются четыре связи (три межуровневые и одна одноуровневая) между объектами *F*3, *F*2, *F*1, *F*4, *F*5. При этом в блоках *F*2 и *F*1 появляется по одному новому выходу, в блоке *F*4 – новый вход. Именование сгенерированных входов-выходов ФБ осуществляется на основе имени связи или имен ролей связи.



а) *б*)

Рис. 2.6. Виды связей по локализации:
а – связи в диаграмме объектов; *б* – связи в системе ФБ

В качестве второго, более сложного примера рассмотрим реализацию транзитных связей между экземплярами ФБ типов *G* и *H* на *UML-FB*-диаграмме (см. рис. 2.4). Например, для реализации транзитной связи между экземплярами ФБ *F*1.*F*3.*F*4.*F*5:*G* и *F*2.*F*7:*H* требуется пять локальных связей. Каждый из ФБ: *F*1.*F*3.*F*4:*E*, *F*1.*F*3:*D* и *F*1:*C* – требует нового выхода, а блок *F2*:*K* требует нового входа. Именование сгенерированных входов и выходов было произведено в соответствии с именами связей или именами ролей связей. На рис. 2.7 приведена система ФБ, соответствующая диаграмме *UML-FB*, приведенной на рис. 2.4. Как можно заметить, одной «событийной» связи (со стереотипом *EVENT*) на данной *UML-FB*-диаграмме соответствует 14 событийных связей в системе ФБ.

Комментарии к состояниям в диаграмме состояний

Комментарии к состояниям в диаграмме состояний служат для представления алгоритмов *EC*-акций.

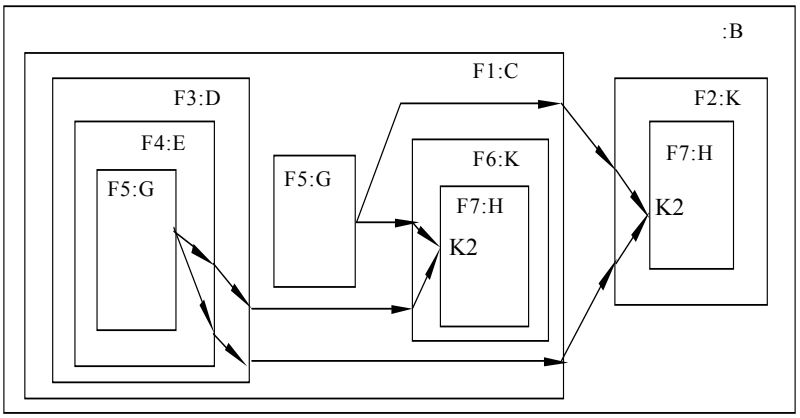


Рис. 2.7. Система ФБ,
соответствующая *UML-FB*-диаграмме на рис. 2.4

Иллюстративный пример

Проиллюстрируем описательные возможности языка *UML-FB*. В качестве примера управляющей системы, построенной на основе IEC 61499, рассмотрим пример из Приложения 2 части 2 стандарта IEC 61499 [164].

Приведенная производственная система предназначена для сортировки продукции (например, апельсинов). Объект управления включает конвейер, бункер для приема бракованной продукции, датчик наличия и цвета и пневматический выталкиватель, расположенный на следующей позиции за датчиком. При обнаружении бракованного продукта (зеленого цвета) поршень выталкивает его с конвейера в бункер. На рис. 2.8 схематично представлена вся система, включающая как управляющую часть, так и объект управления. Управляющая система состоит из приложения, распределенного по двум устройствам и двум ресурсам, причем каждое устройство включает только один ресурс. Каждый из ресурсов содержит по одному коммуникационному СИФБ для передачи данных через сеть. Приложение состоит из СИФБ *SENSOR* и *ACTUATOR*, осуществляющих низкоуровневое взаимодействие с датчиком и выталкивателем соответственно, а также базисного ФБ *GATE*, представляющего логику работы приложения (рис. 2.9). Алгоритм *REQ*, выполняемый в состоянии *REQ* этого ФБ, включает один оператор *OUT* := (*IN*1 & *IN*2).

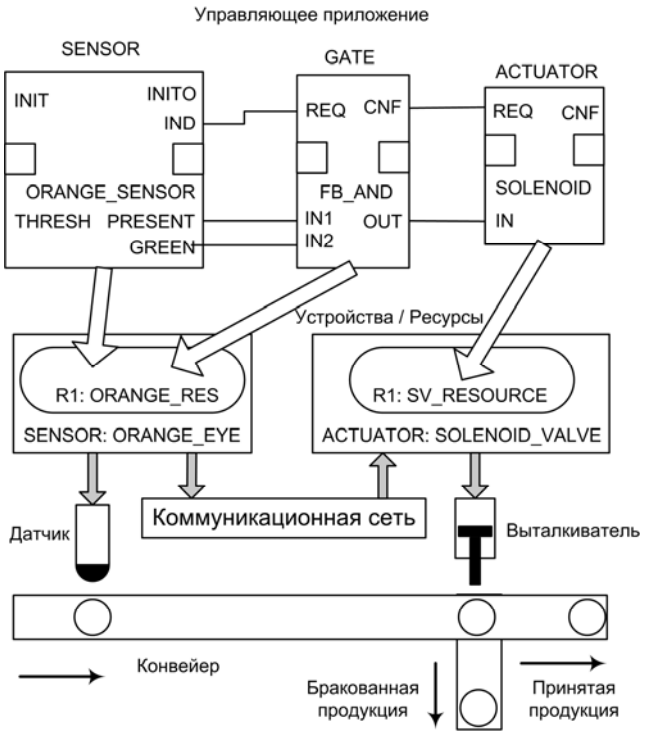


Рис. 2.8. Система сортировки продукции

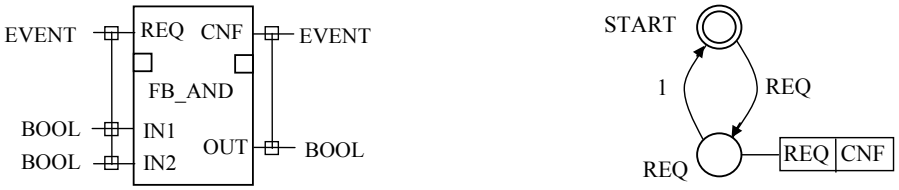


Рис. 2.9. Базисный ФБ, выполняющий операцию конъюнкцию
(интерфейс – слева, диаграмма ECC – справа)

Тип СИФБ *SOLENOID* для взаимодействия с соленоидным приводом представлен на рис. 2.10.

Рис. 2.10. СИФБ *SOLENOID*:
интерфейс (слева) и временная последовательность работы (справа)

На рис. 2.11 приведена диаграмма классов управляющей системы в целом, включающей управляющее приложение, ресурсы и устройства. На рис. 2.12 показана диаграмма кооперации, определяющая связи в управляющем приложении, а на рис. 2.13 – диаграмма состояний, представляющая диаграмму *ECC* блока *FB_AND*.

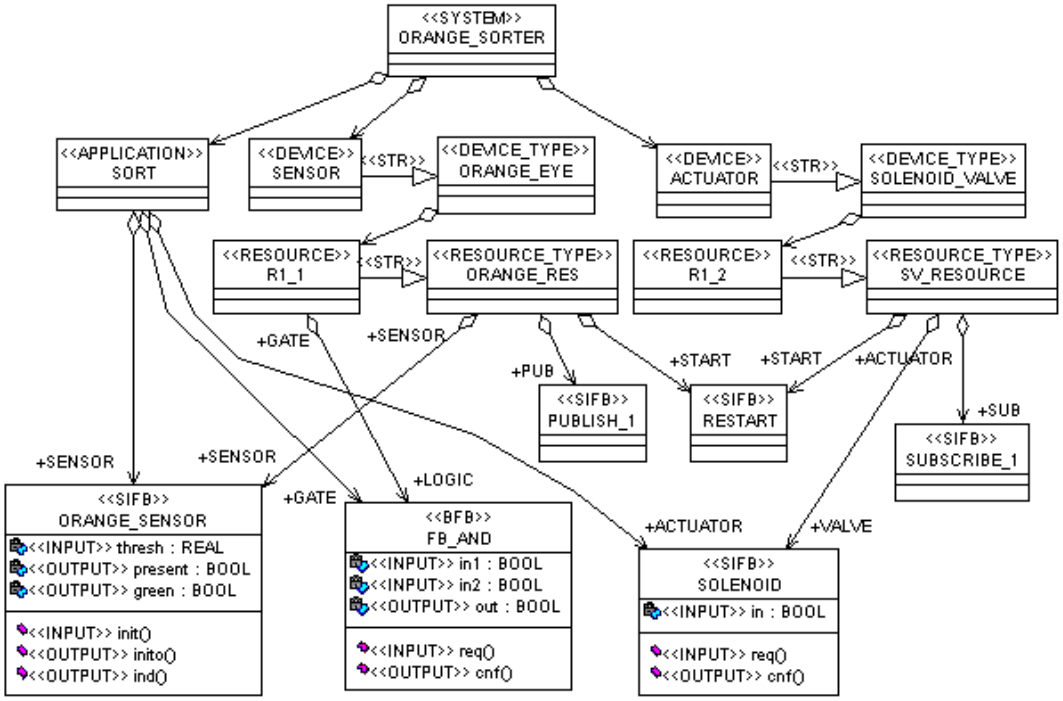


Рис. 2.11. Диаграмма классов системы управления в целом

Рис. 2.12. Диаграмма кооперации,
определяющая связи в управляющем приложении

Рис. 2.13. Диаграмма состояний,
представляющая диаграмму *ECC* блока *FB_AND*

2.2. Разработка UML-FB-спецификации производственной системы FESTO

В данном разделе представлен пример спецификации системы управления производственной системой *FESTO* с использованием языка *UML-FB*. Результатом проектирования является выполнимая спецификация в виде функциональных блоков, полученная в результате трансформации *UML-FB*-спецификации по определенным правилам.

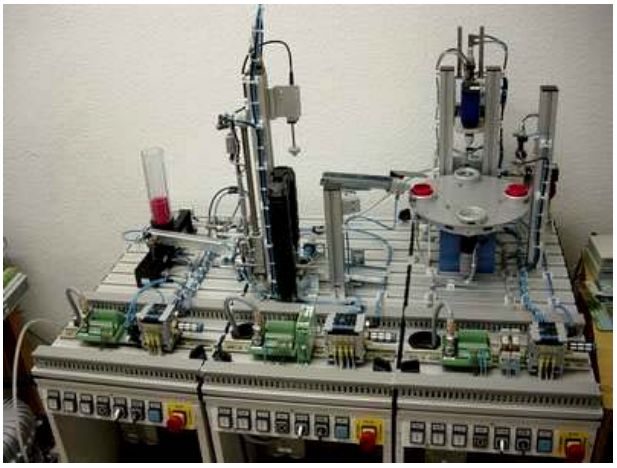


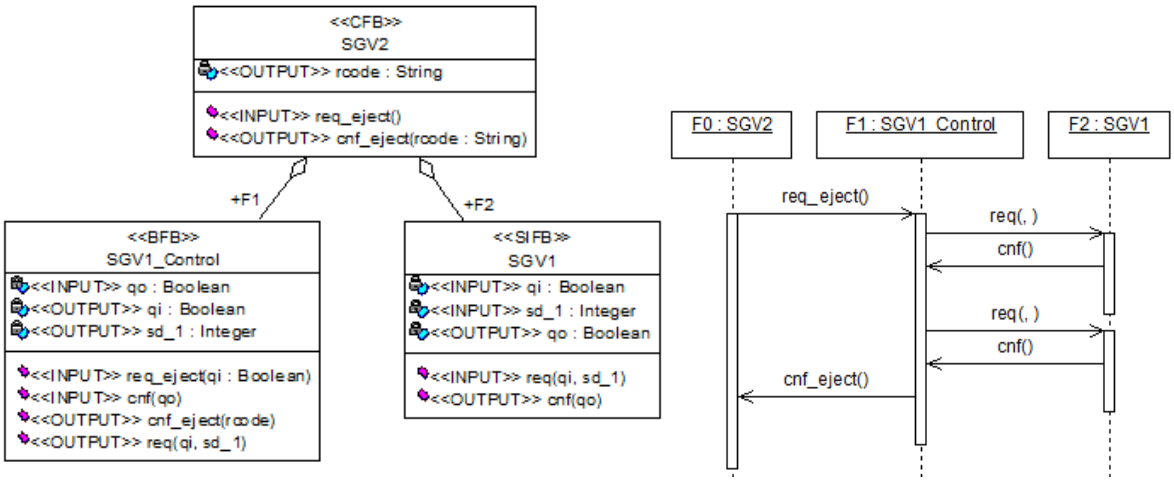
Рис. 2.14. Производственная система FESTO

Производственная система *FESTO* [114] предназначена для изготовления некоторых законченных деталей из поступающих на ее вход заготовок и состоит из трех станций – распределительной, тестирующей и обрабатывающей (рис. 2.14). Проектирование системы управления в рамках IEC 61499 состоит из двух основных этапов: логического и физического. Целью логического этапа является раз-

работка управляющего приложения в виде сети ФБ. На этапе физического проектирования определяется конфигурация системы и устройств, а также размещение ФБ на ресурсах и устройствах системы.

Для проектирования управляющего приложения использовалась методология объектно ориентированного и структурного проектирования. Управляемое устройство (N + 1)-уровня представляется как пара, состоящая из одного (или более) управляемого устройства (N)-уровня и некоторого управления (N)-уровня. Виртуальное устройство (N)-уровня предоставляет сервис более высокого уровня по сравнению с виртуальным устройством (N – 1)-уровня. Аппаратное устройство считается устройством уровня 0. Виртуальное устройство уровня 1 является устройством уровня потока событий и играет особую роль при проектировании системы. Это самое низкоуровневое программное представление устройства (в соответствии с IEC 61499), на котором держится иерархическая система управления. Обычно виртуальные устройства уровня 1 – это СИФБ.

На рис. 2.15 приведен пример диаграмм классов и последовательности выталкивателя уровня 2, разработанного с использованием метода *восходящего проектирования*.



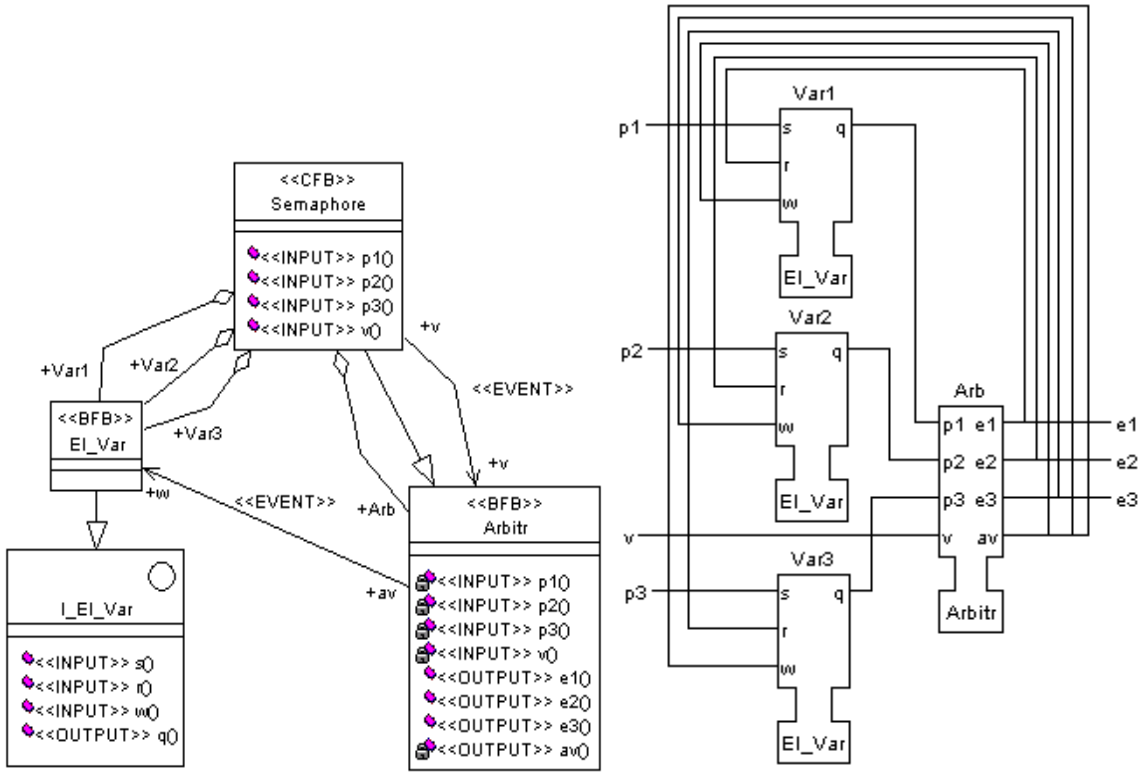
а) *б*)

Рис. 2.15. Выталкиватель уровня 2:
а – диаграмма классов; *б* – диаграмма последовательности

Выталкиватель *SGV*2 (уровня 2) имеет событийные вход *req_eject* – «вытолкнуть деталь» и выход *cnf_eject* – «деталь вытолкнута», а также информационный выход *rcode* – «код завершения операции». Априори заданным является СИФБ *SGV*1, представляющий выталкиватель уровня 1, поставляемый производителем оборудования. Как видно из диаграммы последовательности

(см. рис. 2.15,*б*), выталкивание детали является двухфазным. На первой фазе поршень выталкивателя движется вперед, а на второй – назад.

Для проектирования отдельных, чисто программных компонентов, входящих в состав станций, использовалось *нисходящее проектирование*. Например, при проектировании двоичного семафора [40] сначала был разработан *CFB*-класс *Semaphore*, детализированный в дальнейшем с помощью *BFB*-классов *EI_Var и Arbitr* (рис. 2.16,*а*).



а) *б*)

Рис. 2.16. Двоичный семафор:
а – диаграмма классов; *б* – сеть ФБ

В *CFB*-классе *Semaphore* используются следующие событийные входы: *p*1, *p*2 и *p*3 – запрос закрытия семафора от клиентов 1, 2 и 3 соответственно; *v* –запрос открытия семафора, а также событийные выходы *e*1, *e*2 и *e*3 – подтверждения закрытия семафоров для клиентов 1, 2 и 3 соответственно.

Пример, представленный на рис. 2.16, является иллюстративным также в плане демонстрации использования различных видов отношений в диаграмме классов. Функциональный блок *Semaphore* включает (агрегирует) три экземпляра (*Var*1, *Var*2, *Var*3) ФБ типа *EI_Var*. Одноуровневая связь «*av*→*w*» между классами *Arbitr* и *EI_Var* предполагает, что событийный выход *av* блока *Arb* будет

связан с событийными входами *w* блоков *Var*1,*Var*2 и *Var*3. Межуровневая связь «*v*→*v*» со стереотипом *EVENT* между классами *Semaphore* и *Arbitr* фактически определяет единственную связь между входом *v* оболочки (капсулы) любого экземпляра ФБ типа *Semaphore* и одноименным входом вложенного в него ФБ *Arb* типа *Arbitr*. На данной диаграмме классов из ФБ *Arbitr* в ФБ *Semaphore* наследуются событийные выходы *e*1, *e*2, *e*3. Блок *EI_Var* наследует интерфейс *I_EI_Var*. Сеть ФБ, соответствующая данной диаграмме классов и некоторой диаграмме кооперации, дополняющей уже определенные связи, представлена на рис. 2.16,*б*. Следует заметить, что более подробно правила преобразования *UML-FB*-описаний в системы ФБ представлены в подразделе 2.3 ниже.

Основу конфигурации приложения, осуществляющего управление производственной системой *FESTO*, составляют три субприложения: *Distribution_Control*, *Testing_Control* и *Processing_Control*, используемые для управления распределительной, тестирующей и обрабатывающей станциями соответственно (рис. 2.17).

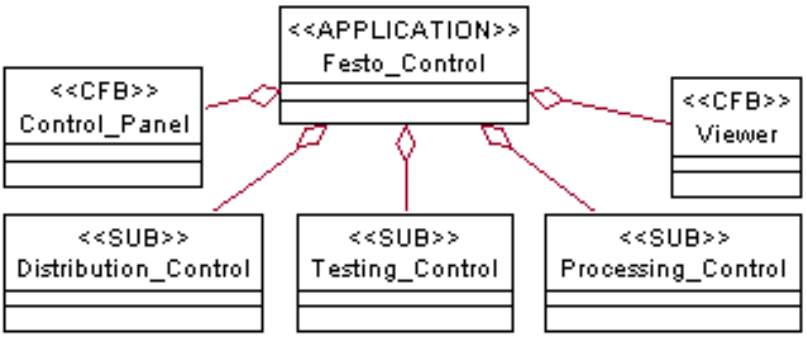


Рис. 2.17. Диаграмма классов для конфигурации приложения

Составной ФБ *Control_Panel* служит для организации и представления человеко-машинного интерфейса в виде панели управления, а блок *Viewer* – для визуализации текущего состояния производственной системы на экране монитора.

Субприложение *Distribution_Control* детализировано на рис. 2.18. Используемые на данной диаграмме классы: *Magazine1* и *Feed_Magazine* – приемный магазин на уровнях 1 и 2 соответственно; *FM_Control* – управление магазином на уровне 1; *SGV*1 и *SGV*2 – выталкиватель детали на уровнях 1 и 2 соответственно; *SGV*1_ *Control* – управление выталкивателем на уровне 1; *Automoving_Control* – управление автоматическим переносом шарнирного рычага с распределительной станции на тестирующую; *Transfer* – виртуальное устройство переноса деталей с распределительной станции на тес-

тирующую; *Swivel*1 – шарнирный рычаг; *Sucker*1 – присоска; *Transfer_Control* – управление устройством переноса деталей; *Semaphore* – двоичный семафор; *EI_Var* – переменная событийного входа (*EI*-переменная); *Arbitr* – арбитр.

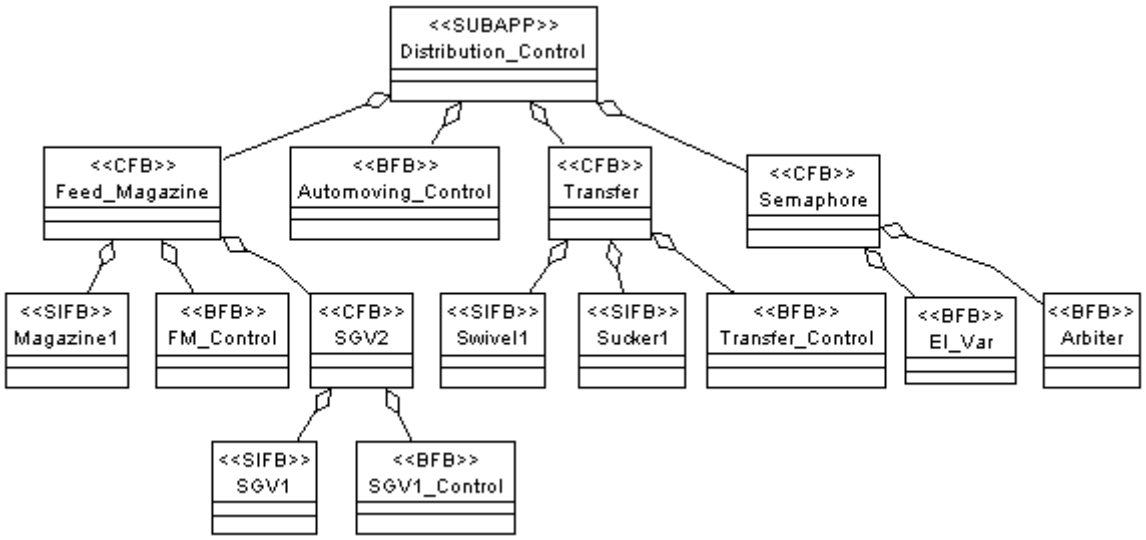


Рис. 2.18. Диаграмма классов для субприложения *Distribution_Control*

Субприложение *Testing_Control* детализировано на рис. 2.19. Используемые на данной диаграмме классы: *Detector*1 – детектор цвета и материала детали; *SGV*1 и *SGV2* – выталкиватель детали на уровнях 1 и 2 соответственно; *SGV1_Control* – управление выталкивателем на уровне 1; *Lift1* – лифт; *Measurer*1 и *Measurer*2 – измеритель высоты детали на уровнях 1 и 2 соответственно; *Measurer1_Control* – управление измерителем высоты детали на уровне 1; *Slide*1, *Slide*2 и *Slide_Buffer* – скользящий желоб на уровнях 1, 2 и 3 соответственно; *Slide*1*_Control* – управление скользящим желобом на уровне 1; *E_DELAY* – таймер; *Binary_Semaphore* – двоичный семафор для взаимного исключения; *EI_Var* – переменная событийного входа (*EI*-переменная); *Arbiter* – арбитр; *Counting_Semaphore* – считающий семафор для подсчета заполненных (*Full*) и пустых (*Empty*) ячеек в буфере и организации соответствующих блокировок; *TS_Control* – управление тестирующей станцией.

Размещение приложения по ресурсам и устройствам системы является самостоятельной нетривиальной задачей, решению которой, в частности, посвящена работа [105]. Можно отметить лишь один из принципов размещения, согласно которому на отдельных устройствах следует размещать те части приложения, которые можно выполнять параллельно.

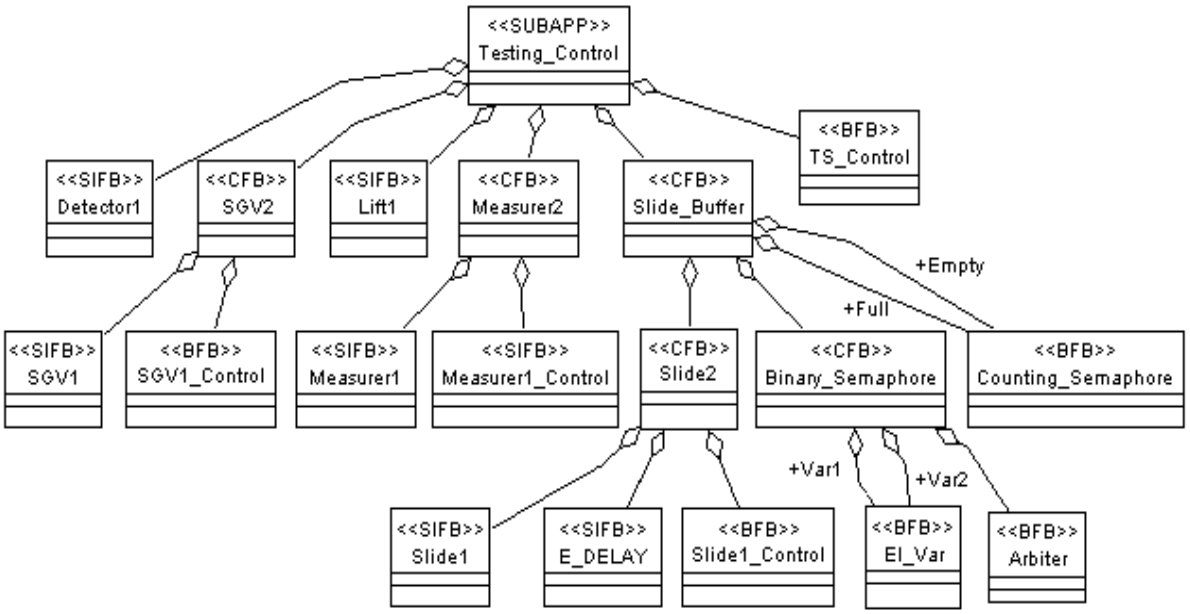


Рис. 2.19. Диаграмма классов для субприложения *Testing_Control*

Система управления *FESTO* в целом строится как децентрализованная система, состоящая из взаимодействующих систем управления станциями. Для организации взаимодействий между станциями используются типовые задачи синхронизации – задача взаимного исключения и задача «производитель – потребителъ».

Конфигурация системы в соответствии с IEC 61499 включает конфигурации связанных с ней устройств и приложений, включая размещение экземпляров ФБ приложения на ресурсах, связанных с устройствами. На рис. 2.20 представлен пример конфигурации системы управления *FESTO* в форме диаграммы классов *UML-FB*.

В состав данной системы входят пять устройств. Устройства *Control*1, *Control*2 и *Control*3 типа *RMT_DEV* («удаленное устройство») [134] предназначены для управления распределительной, тестирующей и обрабатывающей станцией соответственно. На устройствах *HMI* и *VIEW* типа *FRAME_DEVICE* [134] размещены фрагменты приложения, ответственные за человеко-машинный интерфейс и визуальное отображение объекта управления соответственно. Как видно из диаграммы классов, в системе *FESTO* выполняется управляющее приложение *Festo_Control*, приведенное на рис. 2.17. Следует отметить, что представленные устройства наследуют не только атрибуты соответствующих родительских классов, помеченных стереотипом *DEVICE_TYPE*, но и их внутреннюю структуру.

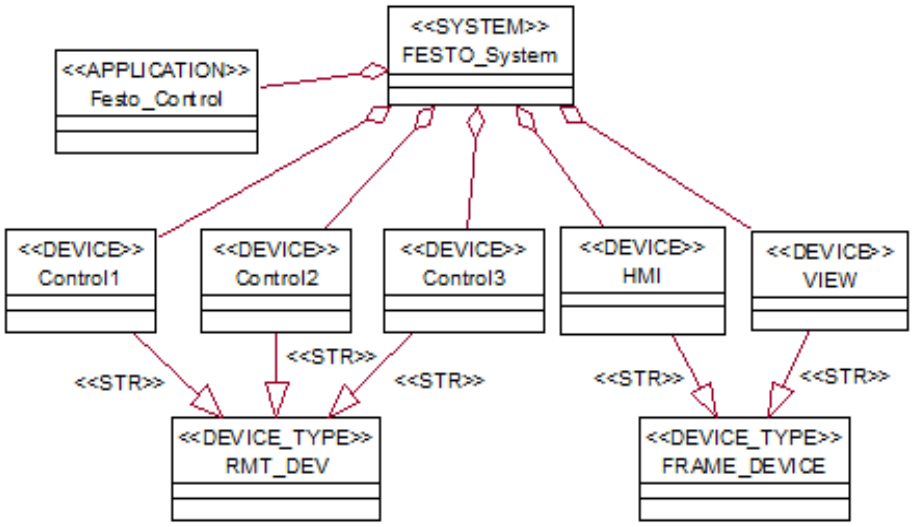


Рис. 2.20. Диаграмма классов
для конфигурации системы управления *FESTO*

На рис. 2.21 в качестве примера приведена (за исключением некоторых деталей) конфигурация устройства *Control*1.

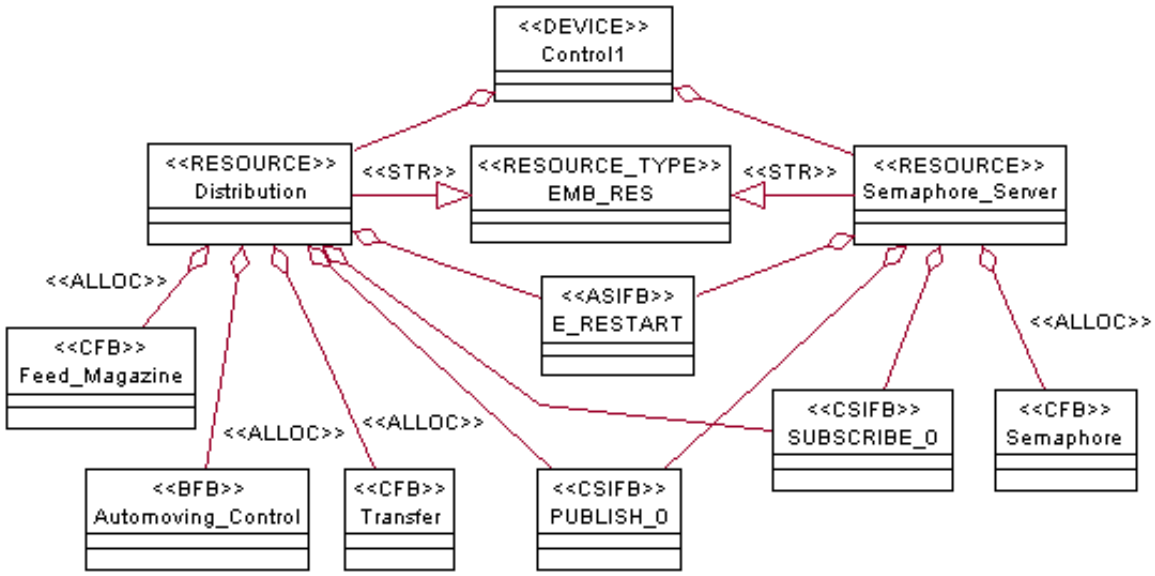


Рис. 2.21. Диаграмма классов
для конфигурации устройства *Control*1

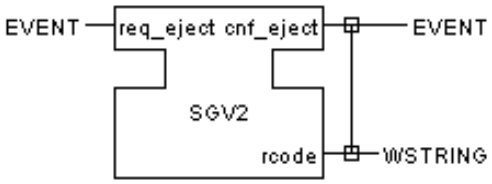
Данное устройство включает ресурсы *Distribution* и *Semaphore_Server* типа *EMB_RES*. На ресурсе *Semaphore_Server* размещен сервер «Двоичный семафор», используемый совместно распределительной и тестирующей станциями. Взаимодействие данного ресурса с другими ресурсами осуществляется через сеть с помощью ком-

муникационных ФБ *PUBLISH_0* и *SUBSCRIBE_0* [134]. На ресурсе *Distribution* размещены ФБ, ответственные за непосредственное управление устройствами распределительной станции. Размещение на ресурсе блоков, входящих в приложение, производится с помощью агрегирующих связей, помеченных стереотипом *ALLOC*. Следует отметить, что ФБ, входящие в состав субприложения, могут быть размещены на разных ресурсах в системе, в то время как внутренние компоненты составного ФБ должны быть размещены на одном и том же ресурсе.

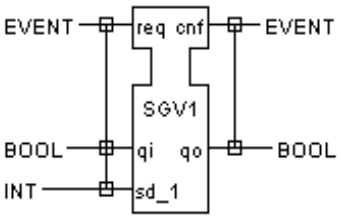
2.3. Трансформация UML-моделей в функциональные блоки

Поскольку полное описание правил трансформации диаграмм *UML-FB* в структуры стандарта IEC 61499 довольно громоздко, ниже представлены лишь некоторые из них. Диаграмма классов используется для генерации типов ФБ, субприложений, адаптеров, ресурсов и устройств, системных конфигураций, а также связей между ФБ.

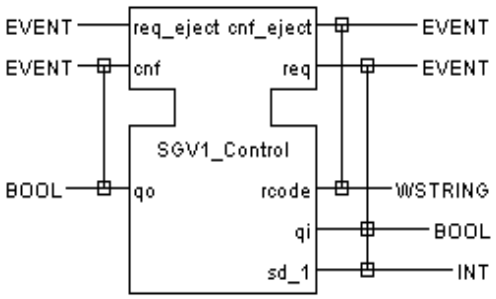
При генерации типов ФБ каждому *BFB*- и *CFB*-классу на диаграмме классов однозначно ставится в соответствие базисный или составной тип ФБ. Классу со стереотипом *SIFB* ставится в соответствие тип СИФБ. Интерфейсы ФБ, сгенерированные по диаграмме классов, изображенной на рис. 2.15,*а*, представлен на рис. 2.22.



а)



б)



в)

Рис. 2.22. Интерфейсы ФБ, соответствующие классам рис. 2.15:
а – выталкиватель уровня 2; *б* – выталкиватель уровня 1;
в – управление выталкивателем уровня 1

Методу класса, имеющему стереотип *INPUT* (*OUTPUT*), ставится в соответствие одноименный событийный вход (выход) ФБ. Методу класса, имеющему параметры, в ФБ ставится в соответствие событийная линия, связанная с информационными линиями, определяемыми этими параметрами, с помощью квалификатора *WITH*.

Атрибуту класса, имеющему стереотип *INPUT* (*OUTPUT*, *INTERNAL*), ставится в соответствие одноименная входная (выходная, внутренняя) переменная ФБ. Атрибутам класса со стереотипами *SOCKET* (*PLUG*) соответствуют адаптерные входы (сокеты) и выходы (штекеры). Связь между классом на диаграмме классов и объектом на диаграммах взаимодействий производится по именам объектов, указанным в имени роли агрегирующей связи.

Диаграммы взаимодействий используются: 1) для генерации связей в сети ФБ, а также других связей в системе; 2) для генерации временной диаграммы последовательностей, описывающей поведение СИФБ. В первом случае каждой связи в диаграмме последовательности соответствует один событийный канал в сети ФБ и несколько информационных каналов, число которых определяется числом параметров метода, приписанного данной связи. Событийный и информационный каналы определяются между одной и той же парой ФБ. В данном случае под каналом понимается одна связь между ФБ (при использовании в диаграмме последовательности локальной связи) или цепочка связей (при использовании транзитной связи).

Имя метода и имена его параметров однозначно определяют имена событийных и информационных входов-выходов ФБ, через которые проходят генерируемые событийный и информационные каналы.

Пример сети ФБ, представляющей «тело» составного ФБ *SGV2*, сгенерированной на основе диаграммы последовательностей, приведенной на рис. 2.15,*б*, изображен на рис. 2.23. Еще один пример трансформации диаграммы классов в сеть ФБ был приведен ранее на рис. 2.16. Отображение диаграммы состояний в диаграмму *ECC* производится в режиме 1:1, поскольку они очень близки друг другу. Каждому состоянию диаграммы состояний соответствует одноименное состояние *ECC*.

На основе представленных выше правил разработаны прототипы прямого и обратного программных конверторов, осуществляющих с использованием *COM*-интерфейса преобразование *UML*-диаграмм инструментального средства *Rational Rose* в соответствующее *XML*-представление ФБ и, соответственно, обратное преобразование [1, 67]. При разработке данного клиентского приложения

Rational Rose использовался язык *Visual C++* и библиотека классов *Chilkat* для работы с *XML* [100]. Трансформация стандартного представления ФБ в объектно ориентированное представление на основе *UML-FB* способствует реализации систем ФБ с использованием объектно ориентированных языков программирования.

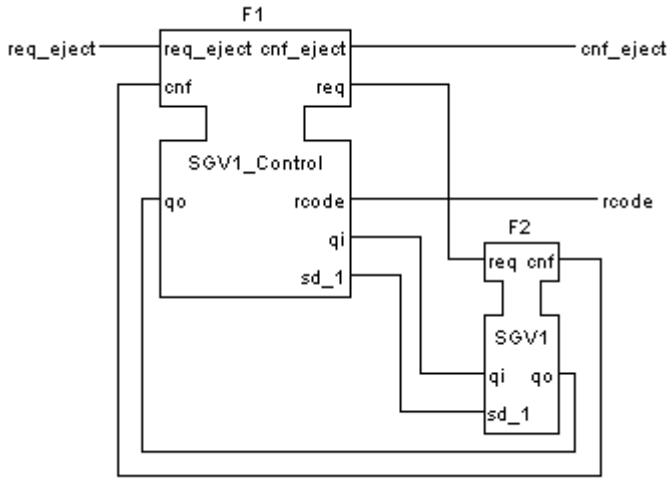


Рис. 2.23. Содержимое составного функционального блока *SGV2*

Следует заметить, что также были разработаны правила трансформации графов для осуществления как прямого преобразования, так и обратного. При этом ФБ и диаграммы *UML-FB* представляются в графовой форме с использованием соответствующих метамоделей.

2.4. Разработка визуальных имитационных моделей мехатронных компонентов

Использование *UML* в качестве основы проектирования инфомехатронных систем создает предпосылки интеграции технологии ФБ с другими видами информационных технологий, такими как базы данных и знаний, *Web*, программными технологиями *Java* и т.д. Ниже представлен пример использования комбинации *UML*-технологии и технологии функциональных блоков для разработки имитационных моделей мехатронных компонентов при использовании широко применяемого шаблона проектирования *Model/View/ Controller* (*MVC*), адаптированного для случая ФБ [101–103].

В случае отсутствия реального мехатронного элемента разрабатывается его программная имитационная модель на основе ФБ (*MV*-компонента), включающая реализацию математической модели элемента (*M*-компоненту) и ее визуальное представление (*V*-компоненту).

В системе управления СИФБ заменяется соответствующим блоком, реализующим *MV*-компоненту, причем замещающий ФБ имеет тот же самый интерфейс, что и заменяемый СИФБ. Во многих случаях содержанием *M*-компоненты являются простые временные задержки. В случае, когда используются элементы, осуществляющие взаимодействие, например, с внешним миром, в *M*-компоненте используются блоки для реализации этого взаимодействия.

Следует отметить, что создание и разработка имитационных моделей мехатронных компонентов полностью вкладывается в технологию объектов автоматизации. Концепция *объектов автоматизации* была предложена в проекте стандарта IEC [82] для интеграции различных парадигм программирования мехатронных устройств, а также поддержке их повторного использования и интероперабельности. Эталонная модель объектов автоматизации не ориентирована на какую-либо конкретную языковую, программную или аппаратную платформу. В соответствии с эталонной моделью объект автоматизации (АО) – это функциональная аппаратная или программная единица для выполнения функций автоматизации и управления. Аппаратная единица может быть каким-либо механизмом, в то время как программная единица может представлять различные модели устройств, интерфейсы, описания и алгоритмы для управления.

Имитационная модель выталкивателя

Структура визуальной имитационной модели выталкивателя в виде диаграммы классов представлена на рис. 2.24.

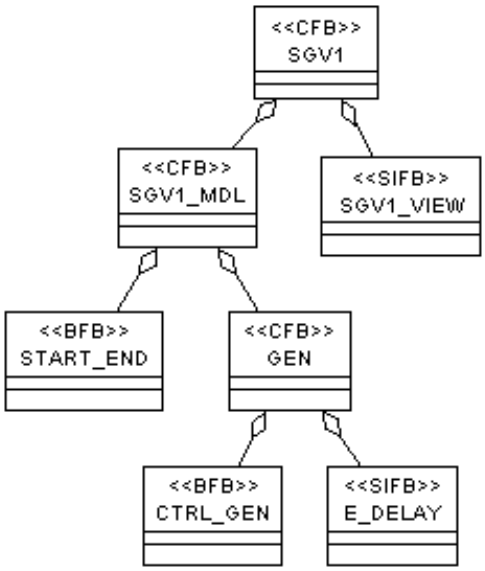


Рис. 2.24. Диаграмма классов для визуальной имитационной модели выталкивателя

В данном случае составной ФБ *SGV*1, представляющий *MV*-компоненту, включает блок *SGV*1_*MDL* – динамическую модель выталкивателя и блок *SGV*1_*VIEW* – визуальную компоненту для отображения выталкивателя на экране монитора.

Блок *SGV*1_*MDL* моделирует прямолинейное равномерное движение поршня как вперед, так и назад. В конце его движения имитируются соответствующие сигналы с датчика. В процессе функционирования данный блок периодически, через заданные интервалы времени, выдает новое местоположение (смещение) поршня выталкивателя.

В свою очередь блок *SGV*1_*MDL* состоит из базисного ФБ *START_END* и составного ФБ *GEN*. Блок *GEN* предназначен для генерации (с заданным временным шагом) последовательности событий конечной длины с выдачей одного, изменяющегося с определенным шагом параметра (например, местоположение поршня).

Блок *GEN* включает блоки *CTRL_GEN* и *E_DELAY*. Базисный блок *CTRL_GEN* используется для управления генератором. Он подсчитывает число сгенерированных событий, проверяет условие окончания процесса генерации, изменяет выходной изменяемый параметр. Блок *E_DELAY* является стандартным СИФБ и реализует временную задержку.

Блок *START_END* предназначен для обработки начала и завершения работы генератора опорных событий. Его основные функции:

- расчет начальных параметров генератора событий;
- запуск генератора событий;
- обработка окончания работы генератора событий и выдача соответствующих событий об изменении статуса выталкивателя.

Визуально выталкиватель представляется в виде цилиндра, в котором ходит поршень, имеющий наконечник в виде круглой шайбы. Графическое параметризованное изображение выталкивателя представлено на рис. 2.25.

Интерфейс СИФБ, определяющий визуальный компонент для отображения выталкивателя на экране, представлен на рис. 2.26.

На рис. 2.27 на примере разработки визуального компонента для выталкивателя отражены некоторые технологические аспекты соединения парадигм, основанных на ФБ и объектно ориентированных языках программирования, в рамках языка *UML-FB*. При этом язык *UML-FB* расширяется с использованием механизма стереотипов.

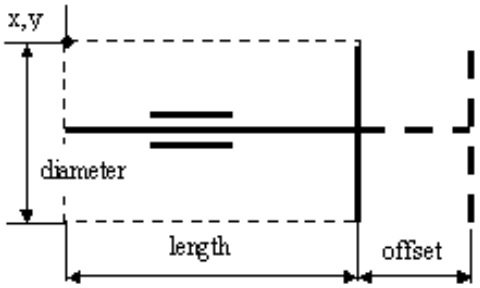


Рис. 2.25. Графическое изображение выталкивателя

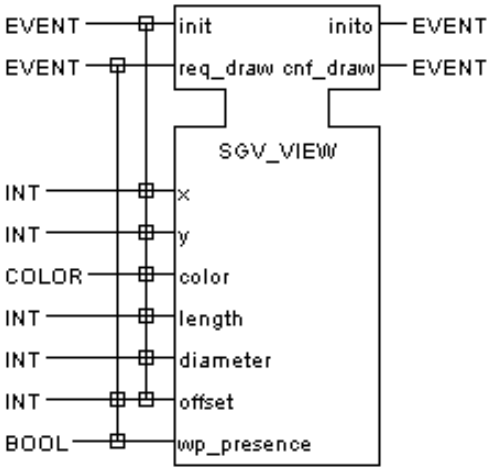


Рис. 2.26. Интерфейс СИФБ для визуализации выталкивателя

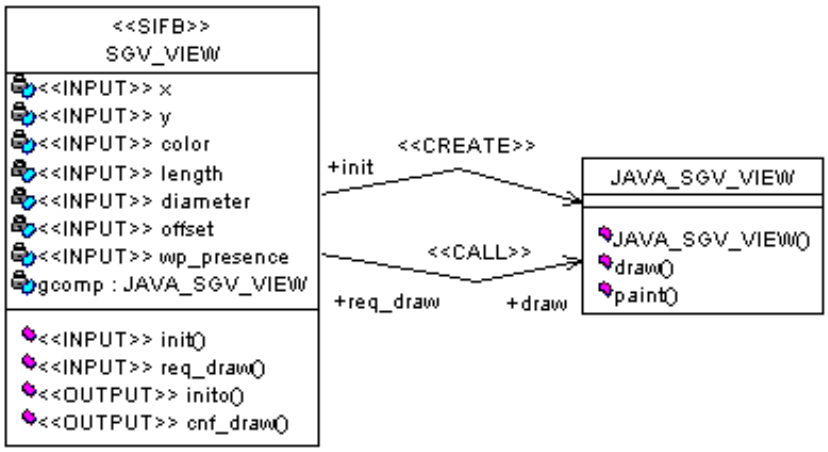


Рис. 2.27. Взаимосвязь *UML-FB*-класса и *Java*-класса для визуализации выталкивателя

В общем случае в инструментальной среде проектирования функциональный блок представляет собой некоторый программный модуль, реализация которого определена для данной конкретной среды. Предположим, что ФБ представляется в виде класса языка программирования *Java*, как это принято в *FBDK* [134]. В этом случае методы и свойства без стереотипа, определенные в классе *UML-FB*, будут представлять методы и свойства программного класса, реализующего данный ФБ в инструментальной системе. Например, в СИФБ-классе *SGV_VIEW* определено свойство *gcomp*, имеющее тип программного класса *JAVA_SGV_VIEW*. Входу ФБ (иначе, методу со стереотипом *INPUT* в классе *UML-FB*) может быть поставлен в соответствие некоторый метод в программном классе. Для этого вводится ассоциативная связь со стереотипом *CALL* между соответствующими классами, причем роль связи со стороны класса *UML-*

FB определяется как имя входа ФБ, а роль связи со стороны программного класса – как имя соответствующего метода в этом классе.

Например, на приведенной на рис. 2.27 диаграмме классов одна ассоциативная связь ставит в соответствие входу *req_draw* блока *SGV_VIEW* метод *draw* программного класса *JAVA_SGV_VIEW*, осуществляющего установку параметров изображения и вывода его на экран с помощью метода *paint*().

Для задания отношения порождения между объектами определенных классов вводится ассоциативная связь со стереотипом *CREATE*. При наличии однозначного соответствия между ФБ, его типом, программным классом, представляющим этот тип ФБ и объектом, являющимся экземпляром этого класса, данная связь позволяет обойтись без использования диаграммы взаимодействий. Роль данной связи со стороны класса *UML-FB* определяет имя входа ФБ, наличие сигнала на котором вызывает конструктор ассоциированного программного класса. Например, на рис. 2.27 входу *init* соответствует конструктор класса *JAVA_SGV_VIEW*. Если программный класс используется в нескольких ФБ, то целесообразно создавать программный класс не с помощью класса *UML-FB* и связи со стереотипом *CREATE*, а в некотором инициирующем классе инструментальной системы и использовать отношение наследования для возможности использования методов и свойств этого класса.

По представленным выше *UML-FB*-спецификациям (см. рис. 2.27) *CASE*-средство должно создать некоторые шаблоны языка программирования, удобные для дальнейшей работы. Например, для входа *req_draw* блока *SGV_VIEW* генерируется следующий программный код на языке *Java* (для *FBDK*):

ALGORITHM req_draw IN JAVA :

public void service_req_draw (*boolean qi*)

{gcomp.draw(*x.value*,*y.value*,*length.value*,*diameter.value*,*offset.value*,*wp_presence.value*);*}*

END_ALGORITHM

Имитационная модель магазина

Интерфейс СИФБ, представляющего магазин на уровне потока событий, изображен на рис. 2.28.

На выходе *ind* появляется событие, когда изменяется состояние магазина или подтверждается запрос статуса магазина *req_status*. Выходное событие *ind* сопровождается выдачей состояния магазина в выходной булевой переменной *status*. Магазин имеет два состоя-

ния: «Пуст» и «Не пуст». Изменение статуса магазина возможно только в случае помещения или извлечения детали из магазина.

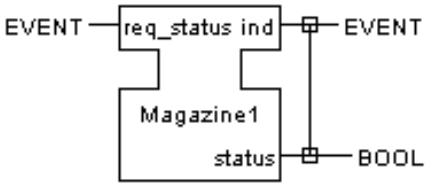


Рис. 2.28. Интерфейс СИФБ, представляющего магазин

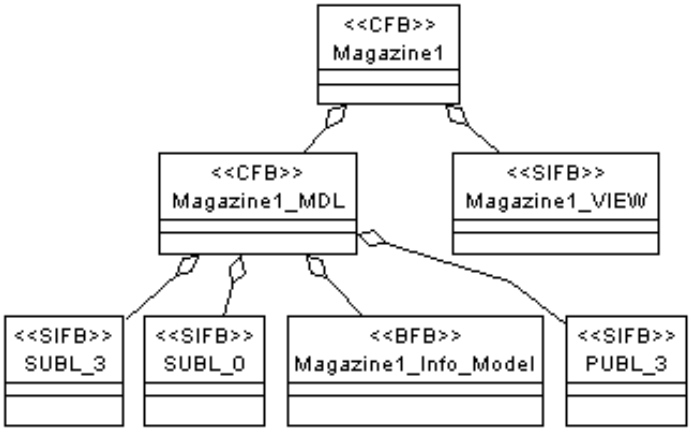


Рис. 2.29. Диаграмма классов для *MV*-составляющей модели магазина

На рис. 2.29 в виде диаграммы классов представлена *MV*-составляющая модели магазина. Модель представлена классом *Magazine*1_*MDL*, а визуальная компонента – классом *Magazine*1_*VIEW*. Сама модель магазина определяет все механические взаимосвязи, а именно: помещение детали в магазин (класс *SUBL*_3), выборку детали из магазина (классы *SUBL*_0 и *PUBL*_3), а также информационную модель магазина в виде определенных структур данных и операций над ними. Блоки типа *SUBL* и *PUBL*, используемые в данной модели, относятся к коммуникационным сервисным интерфейсным функциональным блокам (СИФБ) и реализуют взаимодействие «издатель-подписчик» [134].

Общее визуальное представление системы *FESTO* складывается из отдельных представлений мехатронных компонентов, входящих в ее состав. Переход от имитационной модели системы *FESTO* к реальной системе управления относительно прост. Для этого необходимо заменить *MV*-компоненты на соответствующие СИФБ.

На основе предложенного выше подхода к проектированию визуальных имитационных моделей инфомехатронных систем с использованием языка *UML-FB* и шаблона проектирования *Model/View/Controller* разработана полная визуальная имитационная модель производственной установки *FESTO* [72]. Данная модель реализована на функциональных блоках IEC 61499 в инструментальной среде *FBDK* [134]. Визуальные мехатронные компоненты были разработаны на языке *Java*.

2.5. Графовые шаблонные запросы

При проектировании сложных технических систем значительную часть операций составляют операции поиска и сопоставления различных видов структур, а также операции их преобразования с целью синтеза или верификации. При этом поисковые требования могут оформляться в виде графовых запросов или в терминах путей в графе. Обзор исследований в области систем поиска деревьев и графов можно найти в [210]. В основу систем трансформации структур может быть положен алгебраический или структурно-лигвистический подход на основе графовых грамматик (ГГ) [149]. Эффективность поисково-трансформационного инструментария (для древовидных структур документов) можно наблюдать на примере технологий *XML* (языки *XPath*, *XQuery*, *XSLT*)[128].

Предлагается построение и использование поисковых и трансформационных систем, а также их комбинаций для поддержки проектирования систем промышленной автоматики на основе ФБ. Для поиска в данном случае используются графовые шаблоны (образцы, паттерны). Трансформация структур основана на продукционной модели представления знаний. Поисковый запрос входит в антецедент продукционного правила (продукции). Структурные преобразования, определяемые консеквентом правила, совершаются на основе переписи графов и операций алгебры, определенной над ФБ. Для управления выполнением продукций может использоваться управляющая диаграмма.

Предлагаемый язык графовых шаблонных запросов для систем на основе ФБ является визуальным языком. Описание запроса на данном языке имеет двухуровневую структуру (рис. 2.30).

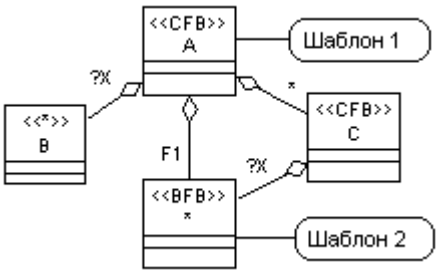


Рис. 2.30. Пример шаблонного графового запроса

Шаблоны верхнего уровня строятся на основе диаграммы классов расширения *UML-FB*. При этом в описании класса могут быть определены событийные и информационные входы и выходы соответствующего ФБ. Шаблоны нижнего уровня адресуются к классам

и присоединяются к ним. Они могут строиться или в терминах языка описания типов ФБ, или на основе диаграммы взаимодействий *UML* (для составного ФБ) и диаграммы состояний *UML* (для базисного ФБ). Шаблон верхнего уровня в запросе один, а шаблонов нижнего уровня может быть несколько. Интерфейс ФБ может быть выражен в запросах обоих уровней. При построении шаблонов нижнего уровня в качестве основы могут использоваться существующие типы ФБ, которые затем модифицируются определенным образом. Например, некоторые элементы могут быть удалены (скрыты) путем их маскирования особыми метками.

В шаблонах допускается использовать символ-заменитель «*» («любой») и переменные. В шаблоне верхнего уровня символ «*» может быть использован вместо имени стереотипа, класса, метода, атрибута, роли дуги, а в шаблоне нижнего уровня – вместо имен событийных и информационных входов и выходов, а также имен типов и экземпляров ФБ. Вершина графа-шаблона, помеченная символом «*», сопоставима с любой вершиной основного графа. В процессе сопоставления шаблона и основного графа производится конкретизация переменных (как в языке Пролог). В результате сопоставления выдается не только соответствие вершин запросного и основного графа, но и значения переменных, при которых возможно вложение шаблона в основной граф. Внешние представления запроса и ФБ, находящихся в базе данных, переводятся в графовые представления для возможности их сопоставления с использованием теории графов.

В основу реализации графовых шаблонных запросов в ряде случаев могут быть положены *XML*-технологии [128]. При этом любая обработка *XML*-документов сводится к обработке структур данных типа «дерево». Наиболее подходящими кандидатами для этих целей являются языки запросов к *XML*-документам (например, язык *XQuery* [71]) и язык трансформаций *XML*-документов – *XSLT* [70].

Не привязываясь к реализации графовых шаблонных запросов, приведем простой пример *XQuery*-запроса к *XML*-представлению ФБ. Запрос «Найти все информационные входы с типом *WSTRING* (например, для ФБ, представленного на рис. 2.31)» на языке *XQuery* в виде выражения *FLWOR*-запроса представляется следующим образом:

```
for $var_inputs in
doc("data/MACH_CTL_PRXY.fbt")/FBType/InterfaceList/InputVars/Var
Declaration
where $var_inputs/@Type='WSTRING'
return $var_inputs
```

Результат выполнения запроса:

<VarDeclaration Name="ID" Type="WSTRING"/>

Для выполнения *XQuery*-запросов может, например, использоваться отдельный интерпретатор *XQuery*-запросов *IPSI-XQ* [165] или интерпретатор *XQuery*-запросов, входящий в систему управления *XML*-базой данных [15], например, СУБД *Exists* [127].

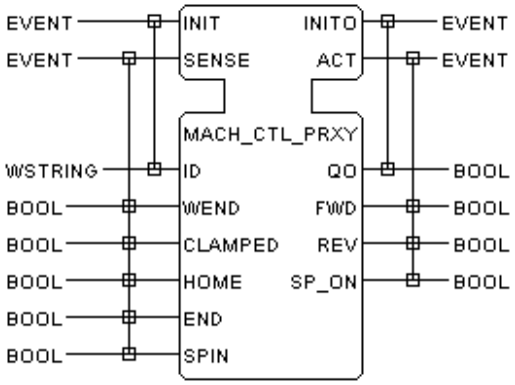


Рис. 2.31. ФБ для иллюстрации *XQuery*-запроса

3. Операционная семантика ФБ

В данном разделе предлагаются два взаимосвязанных подхода к определению формальной семантики ФБ. Первый подход основан на модифицированных распределенных МАС, а второй подход – на СПС. В то же время в силу рамочного характера МАС данная «семантическая» СПС представляется в виде МАС.

Определение семантики ФБ состоит из двух частей: формализации структуры (синтаксиса) и спецификации операционной семантики. Материал будет представляться в указанном порядке. Разделы с 3.1 по 3.4 относятся к первой части. В последующих разделах материал преимущественно будет касаться второй части, но с необходимым представлением синтаксической составляющей.

3.1. Системная конфигурация

Рассмотрим более формально основные артефакты проектирования, определенные в стандарте IEC 61499.

Конфигурация системы *s* (далее – просто система *s*) определяется кортежем:

$$s = (D, DT, dtype, RT, FBT, SAT, A, fbmap, SG, rmap),$$

где $D = \{d_1, d_2, ..., d_{N_D}\}$ – конечное непустое множество конфигураций устройств (далее – просто устройств), входящих в систему *s*; $DT = \{dt_1, dt_2, ..., dt_{N_{DT}}\}$ – конечное, возможно пустое, множество типов устройств, используемых в системе *s*; *dtype*: $D \rightarrow DT \cup \{none\}$ – функция, назначающая устройствам тип устройств; $RT = \{rt_1, rt_2, ..., rt_{N_{RT}}\}$ – конечное, возможно пустое, множество типов ресурсов, используемых в системе *s*; $FBT = \{fbt_1, fbt_2, ..., fbt_{N_{FBT}}\}$ – конечное непустое множество типов ФБ, используемых в системе *s*; $SAT = \{sat_1, sat_2, ..., sat_{N_{SAT}}\}$ – конечное, возможно пустое, множество типов субприложений, используемых в системе *s*; $A = \{a_1, a_2, ..., a_{N_A}\}$ – конечное, возможно пустое, множество приложений системы *s*; *fbmap*: $FB^A \rightarrow FB^S$ – отображение ФБ приложения на ФБ устройств и ресурсов. Здесь под FB^A понимаются все ФБ, входящие в состав приложения и всех (рекурсивно) вложенных в него субприложений; *SG* – множество сегментов локальной вычислительной сети (ЛВС), на которых выполняются приложения; *rmap*: *RS*→*SG* – функция распределения ресурсов системы *s* по сегментам ЛВС. Если все ресурсы некоторого устройства относятся к

одному и тому же сегменту, то считается, что все устройство в целом относится к этому сегменту.

Устройство $d_i \in D$ определяется следующим образом:

$$d_i = (R^{d_i}, RT, rtype^{d_i}, FBN^{d_i}),$$

где $R^{d_i} = \{r_1^{d_i}, r_2^{d_i}, ..., r_{N_{R^{d_i}}}^{d_i}\}$ – конечное непустое множество ресурсов, входящих в состав устройства $d_i \in D$; $rtype^{d_i} : R^{d_i} \to RT \cup \{none\}$ – функция, ставящая в соответствие ресурсам устройства $d_i \in D$ тип ресурса; FBN^{d_i} – сеть ФБ, размещенная на устройстве $d_i \in D$.

Тип устройства $dt_j \in DT$ определяется практически так же, как и устройство:

$$dt_j = (R^{dt_j}, RT, rtype^{dt_j}, FBN^{dt_j}, Par^{dt_j}).$$

В данном случае дополнительный компонент кортежа Par^{dt_j} определяет множество параметров типа устройства $dt_j \in DT$. Каждый параметр есть некоторая константа определенного типа, подаваемая на какой-либо информационный вход сети ФБ FBN^{dt_j}.

Ресурс $r_i \in R$ определяется только своей сетью ФБ:

$$r_i = (FBN^{r_i}).$$

Тип ресурса $rt_j \in RT$ задается двойкой:

$$rt_j = (FBN^{rt_j}, Par^{rt_j}),$$

где Par^{rt_j} – множество параметров типа ресурса $rt_j \in RT$.

Сеть ФБ *FBN* в общем виде определяется кортежем:

$$FBN = (FB, fbtype, FBT, EvConn, DataConn),$$

где *FB* – непустое множество компонентных ФБ сети *FBN*; *fbtype*: $FB \to FBT$ – функция, назначающая компонентным ФБ тип ФБ; *EvConn* – непустое множество событийных связей сети ФБ *FBN*; *DataConn* – непустое множество информационных связей сети ФБ *FBN*.

Все множество типов ФБ делится на три класса (сорта):

$$FBT = BFBT \cup CFBT \cup SIFBT; \quad BFBT \cap CFBT \cap SIFBT = \varnothing,$$

где *BFBT*, *CFBT* и *SIFBT* – множество типов базисных, составных и сервисных интерфейсных ФБ соответственно.

Формальное определение типа ФБ является специфичным для каждого сорта и будет приведено в последующих подразделах.

3.2. Развертывание системной конфигурации

Изначально в соответствии со стандартом IEC 16499 система управления представляется в «свернутом» виде, скорее на уровне типов, чем на уровне экземпляров. Для имитационного моделирования системы управления, ее исследования, а также интерпретации в режиме реального времени (для управления реальными объектами) требуется представление системы на уровне экземпляров. Назовем процесс перехода от системы типов к системе экземпляров в IEC 61499 *развертыванием* системной конфигурации. Развертывание системной конфигурации включает: 1) развертывание ресурсов и устройств; 2) развертывание ФБ-подобных элементов, под которыми будем понимать составные ФБ, субприложения, приложения, а также сети ФБ, используемые на ресурсах и устройствах. Следует отметить, что эти два типа развертываний существенно различаются.

Общий алгоритм развертывания системной конфигурации представлен ниже:

procedure *unfoldSys*(*s*)
  do forall $d_i \in D$
    $dt_j = dtype(d_i)$
    $R_{ext}^{d_i} = R^{d_i} \cup R^{dt_j}$
    $FBN_{ext}^{d_i} = FBN^{d_i} \cup FBN^{dt_j}$
      do forall $r_k \in R_{ext}^{d_i}$
      $rt_m = rtype(r_k)$
      $FBN_{ext}^{r_k} = FBN^{r_k} \cup FBN^{rt_m}$
      $unfoldFB(FBN_{ext}^{r_k})$
      end_forall
  end_forall
end_procedure

Здесь объекты с нижним индексом *ext* есть исходные объекты (без этого индекса), расширенные в результате развертывания. В соответствии с приведенным алгоритмом развертывание устройства сводится к «миграции» всех ресурсов из типа устройства в множество ресурсов собственно устройства, а также объединению сетей ФБ, определенных в устройстве и соответствующем ему типе устройства. Развертывание ресурса включает только добавление сети ФБ, принадлежащей типу ресурса, к сети ФБ, расположенной на са-

мом ресурсе. Дальнейший этап развертывания включает развертывание всех сетей ФБ, расположенных на ресурсах, а также развертывание приложений (если таковые имеются).

Рассмотрим подробнее развертывние ФБ-подобных объектов. Назовем *ссылочным экземпляром* ФБ (субприложения) компонентный ФБ (субприложение), используемый в процессе развертывания. *Развернутым экземпляром* ФБ или субприложения будем называть содержимое (контент), которое соответствует ссылочному экземпляру ФБ или субприложения. Развертывание ФБ-подобного объекта сводится к рекурсивной замене ссылочных экземпляров объектов на соответствующие им развернутые экземпляры. Они получаются путем клонирования описания типа, соответствующего ссылочному объекту. Синтаксически развернутый экземпляр ФБ является копией соответствующего типа. Поэтому в дальнейшем при их описании будем пользоваться введенными обозначениями из описания соответствующих типов. Для определенности речь ниже будет идти о развертывании составных ФБ, однако те же самые рассуждения являются действительными и для всех ФБ-подобных элементов. Сеть ФБ на ресурсе или устройстве можно представить в виде составного ФБ без входов и выходов.

Система экземпляров ФБ полностью определяется деревом иерархии (развернутых) экземпляров ФБ, которое можно определить следующим кортежем:

$$FTree = (M, Aggr, fbtype', idM),$$

где $M = \{M_1, M_2, ..., M_{N_M}\}$ – множество развернутых экземпляров ФБ; $Aggr \subseteq M \times M$ – отношение агрегации; $fbtype'$: $M \rightarrow FBT$ – функция разметки экземпляров ФБ типами ФБ; idM: $M \rightarrow ID^M$ – функция назначения экземплярам ФБ уникальных идентификаторов.

Ниже приводится алгоритм построения дерева иерархии экземпляров. Рекурсивная процедура $unfoldFB(M_i)$ развертывает все входящие в корневой развернутый экземпляр M_i ссылочные экземпляры (любого уровня):

procedure *unfoldFB*(M_i)
if *KindOf*(M_i)$\in$\{*cfb,subappl,appl*\} ***then***
  do forall *fb*$\in$*FBI*(*fbtype'*(M_i))
    newM=InstanceOf(*fbtype*(*fb*))
    Заменить fb на newM
    $M = M \cup \{newM\}$
    $Aggr = Aggrr \cup \{(M_i, newM)\}$
    $fbtype' = fbtype' \cup \{(newM, fbtype(fb))\}$
    $idM = idM \cup \{(newM, newId())\}$

unfoldFB(*newM*)
end_forall
end_if
end_procedure

В приведенной процедуре используются следующие вспомогательные функции. Функция *InstanceOf* формирует экземпляр заданного типа. Функция *KindOf* служит для определения сорта типа заданного экземпляра. Область значений данной функции включает три сорта типа: *cfb* – составной ФБ; *subappl* – субприложение; *appl* – приложение. Функция *FBI* определяет множество ссылочных экземпляров для заданного типа. Функция *NewId* создает новый уникальный идентификатор для созданного развернутого экземпляра.

Замена ссылочного экземпляра развернутым экземпляром производится в три этапа:

1) добавление развернутого экземпляра;

2) встраивание развернутого экземпляра;

3) удаление ссылочного экземпляра. Встраивание развернутого экземпляра производится путем перекоммутации событийных и информационных линий с входов-выходов ссылочного экземпляра на соответствующие входы-выходы развернутого экземпляра. Между интерфейсами развернутого и ссылочного экземпляров должно существовать взаимно однозначное соответствие.

На рис. 3.1 продемонстрировано развертывание развернутого экземпляра M_i. Он был получен из ссылочного экземпляра fb_i. В процессе развертывания M_i ссылочные экземпляры $fb_{i+1}…fb_{i+n}$ заменяются на соответствующие им развернутые экземпляры $M_{i+1}…M_{i+n}$.

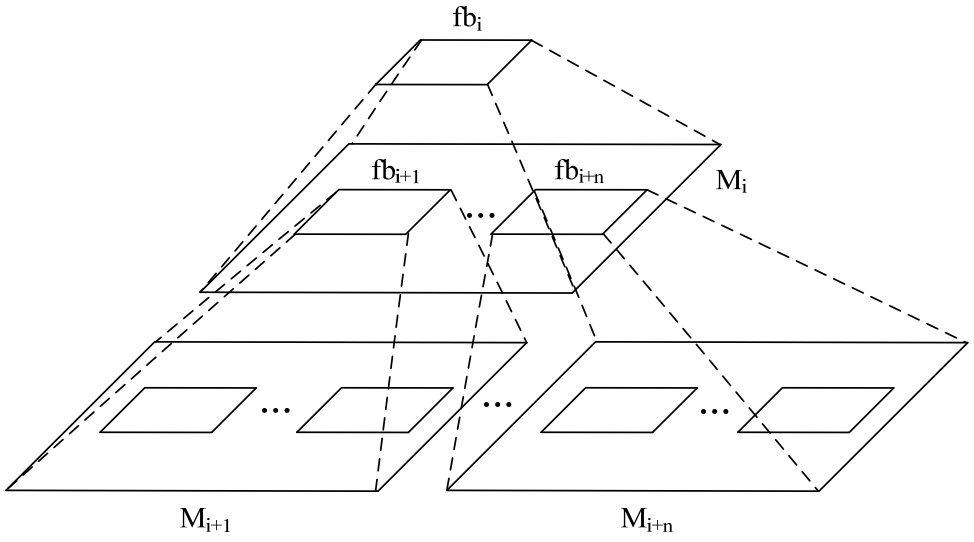


Рис. 3.1. Развертывание экземпляра ФБ

3.3. Буферирование данных

Очевидно, что при моделировании систем ФБ необходимо учитывать семантику синтаксических конструкций. Рассмотрим более подробно передачу данных между ФБ (рис. 3.2).

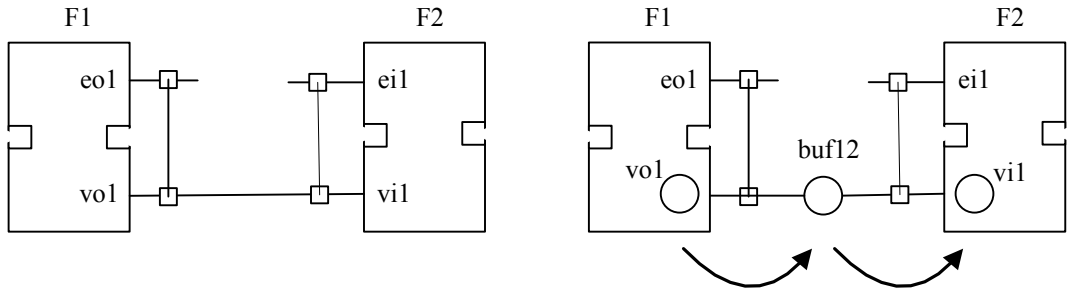


Рис. 3.2. Передача данных между ФБ с буферизацией:
синтаксическое (слева)
и семантическое представления (справа)

Согласно стандарту IEC 61499, части 1 [163] каждому информационному входу и выходу ФБ ставится в соответствие переменная, в которой хранится текущее значение входов и выходов. Информационная связь между выходом одного ФБ и входом другого ФБ, по сути, определяет передачу данных между соответствующими переменными. Однако передача данных между парой ФБ не представляет собой одномоментный (неделимый) акт вследствие того, что выдача и съем данных могут инициироваться разными сигналами. Поэтому в тракт передачи данных необходимо включить буфер (рис. 3.2, справа). На рис. 3.2 (справа) при выдаче сигнала *eo*1 блоком *F*1 данные из выходной переменной *vo*1 переписываются в буфер *buf*12, а при съеме данных в результате обработки сигнала *ei*1 в блоке *F*2 данные из буфера *buf*12 передаются во входную переменную *vi*1. Поскольку буфера данных не определены в Стандарте как самостоятельные элементы (точнее, в Стандарте вообще нет ссылок на них), но в то же время являются необходимыми элементами, то встает вопрос об их размещении. Единственная возможность решения этой проблемы – отнести их к ФБ. Существует два структурных варианта «встраивания» буферов данных в ФБ: 1) соотнести буфер данных с информационным входом ФБ; 2) соотнести буфер данных с информационным выходом ФБ. Недостатком первого варианта является необходимость дублирования данных по всем исходящим из некоторого выхода информационным связям (рис. 3.3, слева). Учитывая топологические ограничения на структу-

ру взаимосвязей ФБ в Стандарте (раздел 2.3.1) [163], согласно которым к информационному входу ФБ может быть подключено не более одной информационной линии, наиболее экономичен и естественен второй вариант (рис. 3.3, справа).

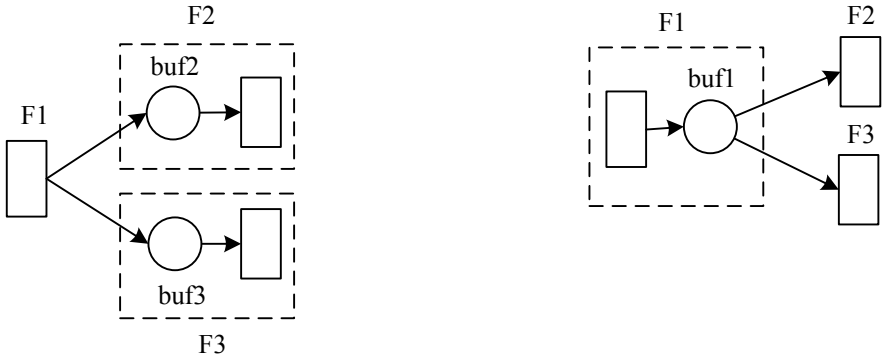


Рис. 3.3. Встраивание буферов данных в ФБ:
a – вариант встраивания «один буфер на один вход ФБ» (слева);
c – вариант встраивания «один буфер на один выход ФБ» (справа)

Используя правило размещения буферов, можно дать следующую семантическую интерпретацию интерфейсным элементам (рис. 3.4):

1) каждому событийному входу (выходу) ФБ соответствует входная (выходная) событийная переменная;

2) каждому информационному входу ФБ соответствует входная переменная;

3) каждому информационному выходу ФБ соответствует выходная переменная и сопряженный с ней однотипный буфер данных;

4) информационным входам и выходам субприложения переменные не сопоставляются;

5) каждой константе на входе ФБ соответствует буфер данных.

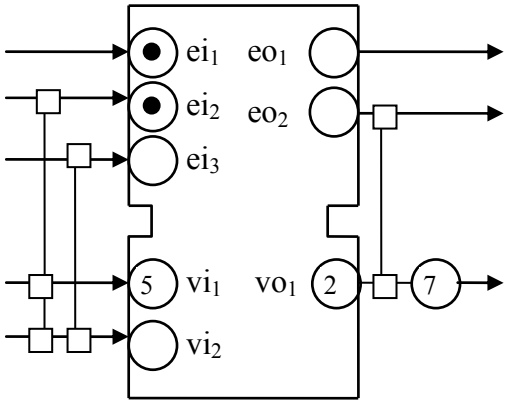


Рис. 3.4. Семантическая модель интерфейса ФБ

3.4. Переход от иерархических структур систем ФБ к одноуровневым

Как видно из рис. 3.1 система ФБ представляет собой иерархическую многоуровневую структуру. Иерархичность системы возникает из-за использования составных ФБ и субприложений. Наличие межуровневых связей зачастую усложняет процесс моделирования и интерпретации систем ФБ. Для избавления от фактора иерархичности предлагается одноуровневое представление (*flat*) систем ФБ, эквивалентное исходной системе. Иногда такое представление называют «*плоским*». Одноуровневое представление могло бы также служить неким каноническим представлением сложных многоуровневых систем ФБ. Понятие одноуровневого представления само по себе не является новым и широко используется в моделировании технических систем (например, [4, 5]).

Однако в случае систем ФБ переход к одноуровневому представлению (*flattening*) сопряжен с определенными трудностями. Это связано с тем, что входной и выходной интерфейсы ФБ имеют определенную логику съема и выдачи данных, что связано с наличием в интерфейсах *WITH*-связей. Для решения этой проблемы предлагается отделить интерфейсную логику ФБ от основной функции, выполняемой ФБ. Для реализации входной и выходной логики ФБ вводится понятие *клапана данных* [48, 117], реализующего передачу данных через интерфейсы.

Клапан данных (КД) представляет программную единицу, осуществляющую при приходе управляющего сигнала копирование данных с информационных входов на выходы. В простейшем случае клапан данных представляет копировщик, управляемый от разветвителя сигнала. На рис. 3.5 приведена структура однопоточного клапана данных с буферизацией. В дальнейшем будем использовать только КД с буферизацией.

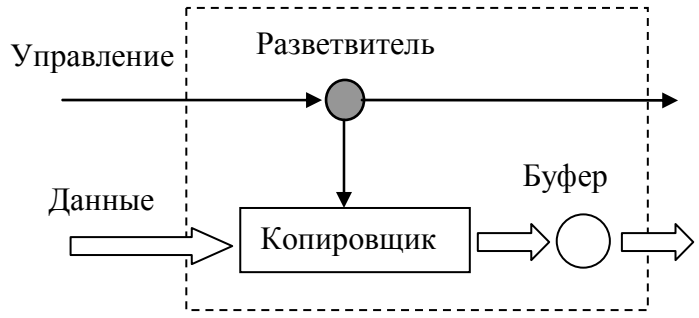
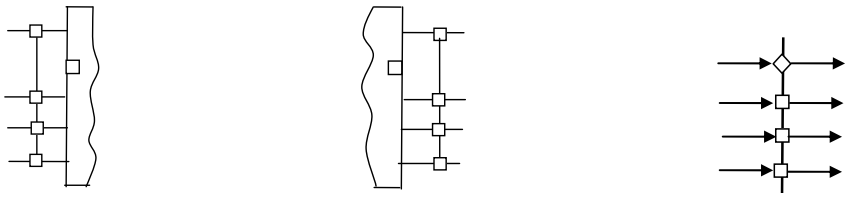


Рис. 3.5. Структура однопоточного клапана данных с буферизацией

В общем случае каждый КД (многопоточный) имеет один событийный вход-выход и несколько информационных входов-выходов. Для представления КД могут использоваться различные нотации [28], одна из которых приведена на рис. 3.6,*в*. Довольно привлекательным является представление КД в виде ФБ. Однако при этом нельзя забывать, что КД являются более низкоуровневыми элементами реализации по сравнению с ФБ. Сети КД используются для моделирования систем интерфейсов ФБ и их представления в одноуровневой структуре ФБ (рис. 3.6). Если КД представляет входной интерфейс ФБ, то назовем его входным, а если выходной – то выходным.

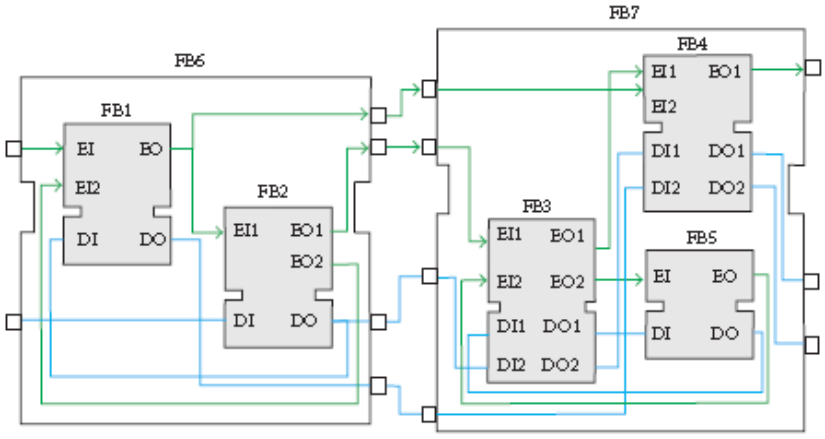


а) *б*) *в*)

Рис. 3.6. Представление входного (*а*) и выходного (*б*) интерфейсов составного ФБ с помощью клапана данных (*в*)

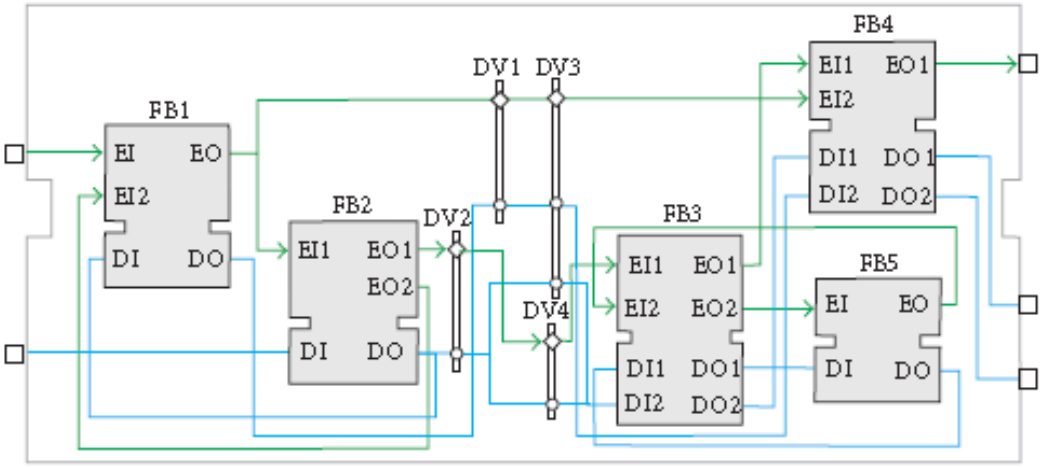
Следует заметить, что при раскрытии субприложений при переходе к одноуровневому представлению понятие КД не используется, поскольку субприложение не имеет интерфейсной логики. В одноуровневом представлении системы ФБ могут присутствовать базисные ФБ, СИФБ, а также КД. Составные ФБ и субприложения в результирующей системе отсутствуют.

Иллюстрация перехода от многоуровневой системы ФБ к одноуровневой представлена на рис. 3.7.



а)

Рис. 3.7. Переход к одноуровневому представлению с использованием клапанов данных (начало):
а – исходная двухуровневая система ФБ



б)

Рис. 3.7. Переход к одноуровневому представлению
с использованием клапанов данных (окончание):
б – результирующая одноуровневая система ФБ

3.5. Модульная формальная модель операционной семантики ФБ

Ниже представлена формальная модель (нотация) для определения операционной семантики ФБ (ФМОСФБ) на основе МАС. Особенности формальной модели: 1) использование переменных состояния и функций значений переменных состояния при определении состояния системы; 2) асинхронность модели; 3) мультиагентность, причем число (неявных) агентов равно числу модулей; 4) наличие общих переменных у модулей; 5) детерминированность (функциональность) модулей; 6) использование явных продукционных правил при представлении программы МАС; 7) особое ограничение на выполнение распределенной МАС. Представление семантики ФБ с использованием ФМОСФБ является годным как для программной реализации, так и для аппаратной, в частности, с использованием языка VHDL [195].

При определении ФМОСФБ будут использоваться следующие соглашения об обозначениях. Пусть Z_A: $A \to Dom(A)$ – функция, назначающая объектам из A значения (из области допустимых значений $Dom(A)$). Тогда $[Z_A]$ будем обозначать множество всех возможных функций Z_A. Число возможных функций из $[Z_A]$ определяется как число размещений с повторениями из n по k и равно n^k, где $k = |A|$, $n = |Dom(A)|$. Наличие взаимно однозначного соответствия между множествами A и B показывается как $A \leftrightarrow B$. Придерживаясь концепции и нотации машин абстрактных состояний Ю. Гуревича,

введем оператор обновления функции значений, обозначив его ←. Данный оператор может быть определен следующим образом:

$$Z_A(a) \leftarrow b \triangleq (Z_A \setminus (a,x)) \cup (a,b),$$

где $a \in A$; $b,x \in Dom(A)$; $Z_A \subseteq A \times Dom(A)$ – график функции. Здесь символ $\triangleq$ читается, как «есть по определению».

По структуре ФМОСФБ представляет совокупность асинхронно работающих (синхронных) модулей:

$$W = (M^1, M^2, ..., M^n).$$

Каждый модуль $M^j \in W$ определяется кортежем (для краткости верхний индекс j для объектов кортежа опущен):

$$M^j = (V, (Dom(v_i))_{v_i \in V}, (T^{v_i})_{v_i \in V}, (p_{v_i})_{v_i \in V}, (Z^0_{v_i})_{v_i \in V}),$$

где $V = \{v_1, v_2, ..., v_m\}$ – множество переменных состояния модуля; $Dom(v_i)$ – область допустимых значений переменной $v_i \in V$; T_{v_i} – функция для вычисления значений переменной состояния $v_i \in V$. Данная функция может быть представлена в глобальном или локальном варианте. В первом случае функция определяется как $T^G_{v_i} : \prod_{v_k \in V} [Z_{v_k}] \to Dom(v_i)$. Поскольку не все переменные влияют на изменение какой-либо другой переменной, то глобальная функция является избыточной и может быть сведена к локальному варианту: $T^L_{v_i} : \prod_{v_k \in H(v_i)} [Z_{v_k}] \to Dom(v_i)$, где $H(v_i) = \{v_{i_1}, v_{i_2}, ..., v_{i_q}\}$ – множество переменных $v_{i_j} \in V$, $j = \overline{1,q}$, которые влияют на изменение переменной $v_i \in V$; $p_{v_i} \triangleq Z_{v_i}(v_i) \leftarrow T_{v_i}(Z_{v_{i_1}}, Z_{v_{i_2}}, ..., Z_{v_{i_q}})$ – правило обновления функции значений переменной $v_i \in V$; $Z^0_{v_i}$ – функция начального значения переменной $v_i \in V$.

Выполнение модуля M^j заключается в одновременном (синхронном) выполнении всех правил $(p_{v_i})_{v_i \in V}$.

Состояние S системы определяется переменными, входящими во все модули W, а также их значениями:

$$S = \prod_{v_k \in V_S} Z_{v_k},$$

где $V_S = \bigcup_{i=1}^{n} V^i$ – множество переменных системы.

Как известно, в теории распределенных МАС порядок выполнения модулей не определяется, только задаются ограничения на этот порядок [58]. Порядок выполнения модулей ФМОСФБ может быть произвольным, но с единственным ограничением – в прогоне ФМОСФБ не должно быть выполнения модулей, не изменяющих текущего состояния. Очевидно, что модули ФМОСФБ должны быть «робастными» в смысле нечувствительности к порядку запуска других модулей. Требование обязательности изменения состояния системы вызвано стремлением избежать бесконечного непродуктивного зацикливания на выполнении «пустых по действиям» модулей. Для обеспечения этого требования предлагаются следующие реализационные схемы (стратегии) выполнения модулей:

1) выполняется тот модуль, у которого изменились входные данные, которые, в свою очередь, могут повлиять на изменение состояния. Из множества переменных V^i модуля M^i выделим общие переменные с другими модулями $V^i_{COM} = V^i \cap \bigcup_{i \neq j} V^j$. В свою очередь, из этих переменных выделим $V^i_{RD} \subseteq V^i_{COM}$ – множество переменных, которые влияют на изменение текущего состояния при выполнении модуля M^i; $V^i_{WR} \subseteq V^i_{COM}$ – множество переменных, значения которых изменяет модуль M^i при своем выполнении. Следует отметить, что общие переменные модулей возникают как следствие наличия вертикальных и горизонтальных связей в моделируемой иерархической системе ФБ. Обозначим $Z^i_{RD_{curr}}$ – упорядоченное множество текущих значений переменных из V^i_{RD}; $Z^i_{RD_{old}}$ – упорядоченное множество значений переменных из V^i_{RD} при предыдущем прогоне. Тогда условие запуска модуля ФМОСФБ формально определяется как $Z^i_{RD_{curr}} \neq Z^i_{RD_{old}}$;

2) после выполнения модуля M^i могут быть выполнены модули, на которые модуль M^i оказывает непосредственное влияние по переменным, т.е. модули из множества $M_{SUCC} = \{M^j \mid V^i_{WR} \cap \cap V^j_{RD} \neq \varnothing, j = \overline{1,n}\}$. Это очевидное следствие принципа локальности изменения глобального состояния при выполнении модуля.

С учетом робастности модуля можно предположить «транзакционный» принцип выполнения модуля ФМОСФБ, согласно кото-

рому модуль выполняется не однократно (как в МАС), а возможно многократно – до достижения неподвижной точки (*fixpoint*), т.е. до тех пор, пока повторные выполнения модуля не будут приводить к изменениям переменных состояния, локализованных в модуле. Использование этого принципа не повлияет на конечный результат, но может быть удобно при программной реализации системы.

Ниже ФМОСФБ будет использована для описания семантики систем ФБ, функционирующих в соответствии с различными моделями выполнения. Правила изменения функций значений переменных, представленные в ФМОСФБ в самом общем виде, будут детализироваться с использованием продукционных правил. В самом простом случае одному правилу изменения функции будет соответствовать одно продукционное правило. Общий вид продукционного правила: $p_t^m : c \Rightarrow a$, где p_t^m – идентификатор правила (или группы правил); c – условие применения правила, a – действия по изменению переменной. Идентификатор правила включает: t – имя изменяемой переменной; m – модификатор идентификатора правила, позволяющий получить дополнительную идентифицирующую информацию о правиле. Модификатор представляется в формате B, C, D или B, C, где B – номер правила среди правил по изменению переменной t; C – идентификатор модуля, в котором локализовано правило; D – идентификатор используемой модели выполнения (для модулей диспетчера). Условия в левых частях правил по изменению одной и той же переменной являются *взаимоисключающими*, поэтому коллизий правил по записи не происходит. Порядок записи правил является несущественным. Для краткости представления однотипных правил может использоваться параметризация правил и группирование параметризованного правила в виде множества.

Для удобства в ряде случаев будем использовать группы приоритетных правил для изменения переменной. Группа правил с приоритетами будет заключаться в угловые скобки: $<p_1: c_1 \Rightarrow a_1; p_2: c_2 \Rightarrow a_2; \ldots p_n: c_n \Rightarrow a_n>$. В процессе интерпретации делается попытка выполнения правила, находящегося ближе к списку, начиная с первого. Если такое правило было найдено и выполнено, то последующие правила из данной группы не выполняются. Следует заметить, что приоритетная группа правил легко может быть преобразована в «бесприоритетную» группу:

$$< p_1 : c_1 \Rightarrow a_1; p_2 : c_2 \wedge \overline{c_1} \Rightarrow a_2; \ldots p_n : c_n \wedge \overline{c_1} \wedge \overline{c_2} \wedge \ldots \wedge \overline{c_{n-1}} \Rightarrow a_n > .$$

3.6. Функционально-структурная организация моделей систем ФБ

Далее семантической моделью (или просто моделью) системы ФБ будем называть модель, построенную в соответствии с нотацией ФМОСФБ. При этом остаются открытыми вопросы выделения модулей, составляющих основу модели всей системы. Под системой ФБ будем понимать систему экземпляров ФБ, корневым элементом которой являются приложение, составной ФБ, субприложение, сети ФБ, расположенные на устройствах или ресурсах. Будем принимать во внимание только базисные и составные ФБ. СИФБ в дальнейшем не рассматриваются, поскольку они представляют собой интерфейс с внешней средой, которая в стандарте никак не определяется.

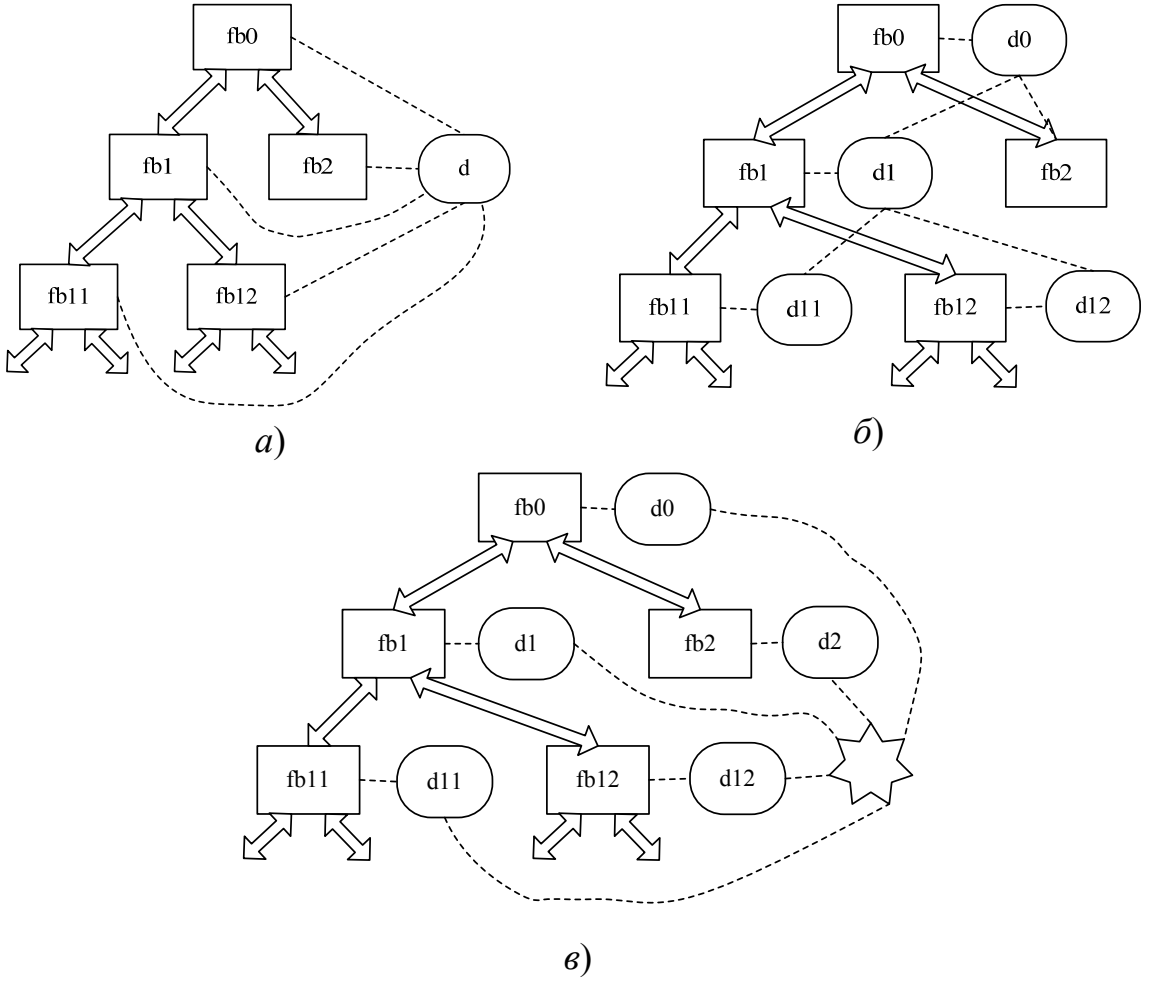
При определении семантики ФБ следует учитывать две составляющие: 1) работу собственно ФБ, неформально описанную в стандарте IEC 61499; 2) работу системы управления выполнением ФБ на ресурсе, обеспечивающую заданную модель выполнения ФБ. Перед разработкой семантики выполнения ФБ следует оценить в целом возможную функционально-структурную организацию формальной модели с целью: 1) выделить инвариантные и изменяемые части с тем, чтобы облегчить описание совокупности семантик ФБ для различных моделей выполнения путем повторного использования инвариантных описаний; 2) использовать полученную функционально-структурную организацию при реализации инструментальных сред выполнения (*run-time*).

На рис. 3.8 на едином примере системы ФБ приведены возможные варианты функционально-структурной организации семантических моделей систем ФБ. На данном рисунке под *fb*0, *fb*1, *fb*11 и *fb*12 понимаются операционные модели составных ФБ или субприложений, а под *fb*2 – операционная модель базисного ФБ. Модель *fb*0 также может представлять приложение в целом. Широкие дуги представляют естественные потоки событий и данных между ФБ смежных уровней, а пунктирные линии – информацию, используемую в управлении выполнением ФБ.

С точки зрения структурной организации самым простым является вариант с общим диспетчером (вариант *a*), однако это, как правило, наиболее сложный вариант по функциональности. В этом случае диспетчер должен хранить и обрабатывать информацию со всех ФБ с учетом их иерархической подчиненности. Кроме того, данный вариант не является масштабируемым и труден в модификации – добавление нового ФБ в систему требует основательной переделки

модели диспетчера. Тем не менее данный вариант является универсальным в том смысле, что подходит для реализации любой модели выполнения ФБ.

В варианте *б* диспетчирование представлено совокупностью иерархически взаимосвязанных диспетчеров. Управляющая информация фактически циркулирует только в этой системе диспетчеров. Все диспетчеры являются простыми, однотипными и минимально связанными с операционными моделями самих ФБ. В принципе, возможно, как это будет показано ниже, естественное слияние операционной модели ФБ с соответствующей ей моделью диспетчера. Вариант *б* допускает простой переход от одной модели выполнения к другой путем замены диспетчера. Однако следует заметить, что данная схема «вкладывается» только в те модели выполнения, в которых запуск ФБ не связан с хронологией выдачи выходных сигналов (например, подходящими являются циклическая и синхронная модели выполнения).



а) *б*)

в)

Рис. 3.8. Функционально-структурная организация моделей систем ФБ:
а – с общим диспетчером; *б* – с иерархически связанными диспетчерами;
в – с независимыми диспетчерами, взаимодействующими через общую память

Вариант *в* определяет систему независимых, вернее, слабосвязанных диспетчеров, взаимодействующих через общую память. В случае оперирования информацией, связанной с представлением иерархической структуры, объем разделяемых данных сильно возрастает и система фактически переходит в разряд сильносвязанных централизованных систем (наподобие варианта *а*) со всеми вытекающими отсюда последствиями.

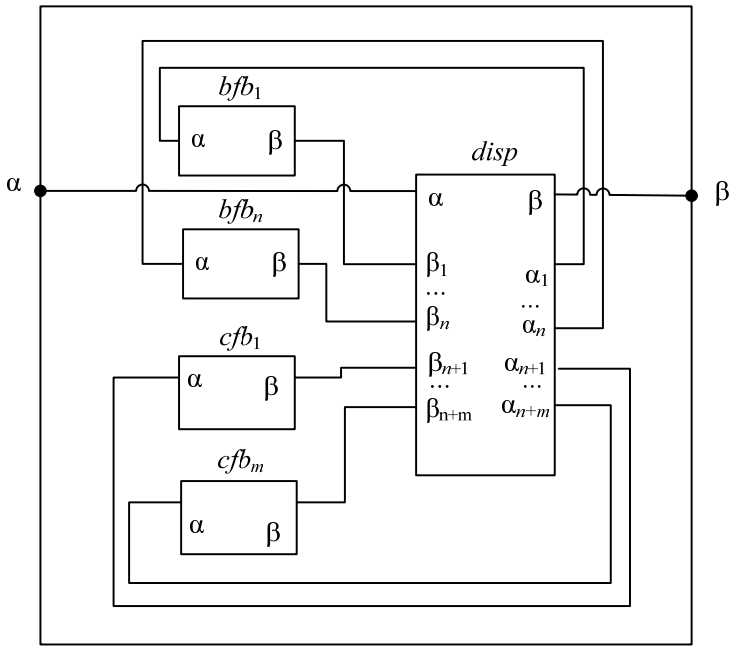
В данном разделе рассматриваются три основные модели выполнения ФБ, принятые сообществом OOONEIDA [189]: циклическая, синхронная и последовательная (на основе последовательной гипотезы). В дальнейшем в рамках синхронной модели будем рассматривать двухтактную схему выполнения, поскольку такая схема является более робастной. В данной модели на первой фазе шага выполнения производится съем данных из внешней среды и выполнение компонентных ФБ, а на второй фазе – передача данных между компонентными ФБ в самой подсистеме и внутри составных компонентных ФБ, а также выдача данных во внешнюю среду.

На рис. 3.9 приведены функциональные структуры операционных моделей составных ФБ (а также субприложений) для циклической и синхронной моделей выполнения, реализованных с использованием второго подхода (рис. 3.8,*б*). На рисунке используются следующие обозначения: bfb_1, … , bfb_n – модели базисных ФБ; cfb_1, … , cfb_m – модели составных ФБ; *disp* – модель диспетчера. Как видно, для удобства и наглядности модель диспетчера включена в состав модели ФБ. На рис. 3.9 приведены только те входы-выходы и связи, которые влияют на порядок выполнения ФБ.

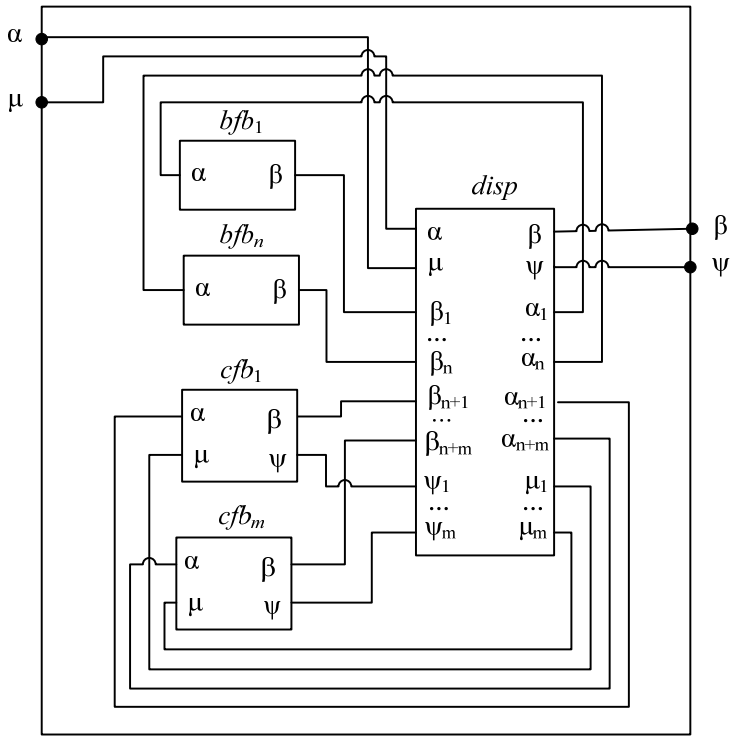
В циклической модели выполнения (см. рис. 3.9,*а*) при получении ФБ стартового сигнала α производятся следующие действия: 1) передвижение сигналов с входов модуля ФБ на входные переменные компонентных ФБ; 2) поочередный запуск всех компонентных ФБ диспетчером; 3) передвижение сигналов между компонентными ФБ и передача сигналов с выходов компонентных ФБ на выходы модуля ФБ; 4) формирование сигнала окончания β.

В синхронной модели выполнения (см. рис. 3.9,*б*) последовательность действий в подсистеме следующая:

Фаза 1 (*по стартовому сигналу* α): 1) передвижение сигналов с входов модуля ФБ на входные переменные компонентных ФБ; 2) запуск первой фазы во всех компонентных ФБ; 3) ожидание завершения первой фазы выполнения во всех компонентных ФБ; 4) формирование сигнала окончания β.



а)



б)

Рис. 3.9. Структура операционных моделей составных ФБ для циклической (*а*) и синхронной (*б*) моделей выполнения

Фаза 2 (*по стартовому сигналу* μ): 5) запуск второй фазы во всех компонентных *составных* ФБ; 6) ожидание завершения второй фазы выполнения во всех компонентных составных ФБ; 7) передвижение сигналов между компонентными ФБ и передача сигналов с выходов компонентных ФБ на выходы модуля ФБ; 8) формирование сигнала окончания ψ.

В предложенной выше модели организации вычислений с использованием иерархической системы диспетчеров в каждом составном ФБ порядок межуровневых передач можно варьировать, изменяя порядок выполнения действий на уровне модуля ФБ. Для того, чтобы обеспечить «правильную» передачу сигналов с блоков верхнего уровня на блоки нижнего уровня и в обратную сторону, в каждом модуле ФБ необходимо производить передвижение сигналов с входов модуля в начале, а выдачу сигналов на выходы – в самом конце работы модуля.

Модель выполнения на основе последовательной гипотезы, несмотря на последовательный характер, является наиболее сложной в логическом плане. В данном случае центральным элементом диспетчирования является очередь событий, общая для всех модулей. Для реализации этой модели используется структура, приведенная на рис. 3.8,*в*.

3.7. Базовая модель базисного функционального блока

3.7.1. Преобразование алгоритмов базисных ФБ

Алгоритмы базисных ФБ представляют простые последовательные алгоритмы, определяемые на языках IEC 61131-3 (кроме языка *SFC*) [1]. Как правило, используются языки *ST* и *LD*. Ниже представлены простые методы преобразования алгоритмов базисных ФБ, предназначенные, с одной стороны, для упрощения их формального представления, а с другой – для их распараллеливания. В последнем случае параллельные участки алгоритма выполняются в едином шаге, что может значительно уменьшить число достижимых состояний при проведении верификации. Для подобного последовательно-параллельного выполнения алгоритма вводится специальный указатель *NI* выполняемого шага инструкций (аналог счетчика команд в архитектуре фон Неймана). Далее для определенности будем рассматривать высокоуровневый процедурный язык *ST*, однако полученные результаты относятся ко всем языкам ПЛК.

Краткая методика преобразования алгоритмов базисных ФБ приведена ниже:

1) преобразовать алгоритмы к виду, содержащему только операторы присваивания, условные операторы и операторы перехода. Как правило, алгоритмы ФБ просты, и для данного преобразования не требуется сложных манипуляций;

2) разбить алгоритм по шагам выполнения. При разбивке учитывается логика выполнения алгоритма. Если, например, один из операторов присваивания модифицирует переменную в правой части другого оператора присваивания, то такие операторы нельзя помещать в один шаг. Кроме того, в один шаг нельзя помещать конфликтные операторы. Два оператора считаются конфликтными, если в одном и том же шаге они могут реально модифицировать одну и ту же переменную. В самом простейшем случае шаг алгоритма будет включать всего один оператор. В каждый шаг добавляются действия по изменению указателя выполняемого шага *NI* (в соответствии с логикой алгоритма). При использовании линейных алгоритмов действия над *NI* сводятся к его инкременту на единицу и могут быть вынесены из описания алгоритмов. Как можно заметить, в преобразуемый алгоритм вводится новая управляющая переменная (*NI*) на правах обычной переменной.

Будем различать шаги двух видов. В шаге первого вида условие изменения связано с одной переменной, используемой в шаге. Структура этого шага в общем случае может быть представлена следующим образом (с использованием БНФ):

IF <Условие 1> THEN <Изменение переменной v1>

…

IF <Условие N> THEN <Изменение переменной vN >

IF <Условие (N+1)> THEN <Изменение счетчика шагов NI>

Следует заметить, что оператор *IF…THEN…ELSE* легко может быть представлен подобным образом, поэтому ниже (на рис. 3.10) будем использовать его в своем «натуральном» виде.

Шаг второго вида имеет единое «сторожевое» условие для всего шага в целом:

IF <Условие> *THEN*

DO

<Изменение переменной 1 >

…

<Изменение переменной v_N >

<Изменение счетчика шагов *NI*>

END

Шаг типа 2 будет использоваться в формальной модели ФБ в виде системы переходов состояний в подразделе 3.14. Следует заметить, что использование шагов типа 2 (незначительно) снижает возможности в параллельном изменении переменных, поскольку

условие вводится для всего шага, а не для каждой переменной в отдельности. Нетрудно перейти от шага типа 2 к шагу типа 1. Обратное неверно.

В качестве примера рассмотрим преобразование алгоритма для подсчета числа положительных (*SP*), отрицательных (*SN*) и нулевых элементов (*SZ*). Для определенности исходный алгоритм представлен на языке *ST*, широко используемом в ПЛК [162]. На рис. 3.10 (слева) приведен исходный алгоритм, содержащий цикл, а (справа) – тот же алгоритм, но уже с разбивкой по шагам выполнения.

```
SZ:=0; SN:=0;
SP:=0;
FOR i=1 TO N
DO
IF a[i]>0
THEN
SP:=SP+1;
ELSIF a[i]<0
THEN
SN:=SN+1;
ELSE
SZ:=SZ+1;
END_IF
END_FOR
```

Номер шага	Действия
1	SZ:=0; SN:=0; SP:=0; i:=1; NI:=NI+1;
2	*IF* a[i]>0 *THEN* SP:=SP+1; *ELSIF* a[i]<0 THEN SN:=SN+1; *ELSE* SZ:=SZ+1; *END_IF* i:=i+1; NI:=NI+1;
3	*IF* i ≤ N *THEN* NI:= 2; *ELSE* NI:=0: *END_IF*

Рис. 3.10. Алгоритм для подсчета сумм положительных, отрицательных и нулевых значений:
слева – с циклом; справа – с разбивкой по шагам выполнения

На шаге 3 сброс указателя шагов в ноль означает конец выполнения алгоритма. Следует отметить, что этап преобразования исходного алгоритма к алгоритму, структурированному по шагам выполнения, легко может быть автоматизирован. Более подробно с вопросами распараллеливания выполнения алгоритмов в синхронных архитектурах можно познакомиться в работе [11].

3.7.2. Определение схемы модели

Модуль (или модель) базисного ФБ (МБФБ) может быть представлен формально следующим образом:

$$M_B = (Synt_B, Sem_B),$$

где $Synt_B$ – синтаксическая часть описания (на уровне абстрактного

синтаксиса); Sem_B – семантическая часть описания. Первая часть определяется исходным описанием ФБ на уровне конкретного синтаксиса, причем для представления ФБ могут использоваться текстовый язык ФБ [163] или XML-описание, определенное в [164]. Первоначальное описание семантики ФБ можно найти в двух источниках: 1) в стандарте IEC 61499 (на естественном языке с использованием диаграмм); 2) в публикациях, посвященных моделям выполнения ФБ. Данные описания, как правило, приводятся на естественном языке с использованием диаграмм.

Синтаксическая часть описания МБФБ определяется четверкой:

$$Synt_B = (Interface, Alg, VV, ECC, Z_B^0),$$

где *Interface* – интерфейс ФБ; $Alg = \{alg_1, alg_2, ..., alg_{N_{Alg}}\}$ – конечное множество (возможно пустое) алгоритмов, выполняемых в базисном ФБ. Каждый алгоритм $alg_i \in Alg$ может быть определен в виде функции вида $alg_i: [Z_V] \rightarrow [Z_{VVO}]$; $VV = \{vv_1, vv_2, ..., vv_{N_{VV}}\}$ – конечное множество (возможно пустое) внутренних переменных, $Dom(VV) = N$, где N – множество целых чисел; VVO – множество внутренних и выходных переменных ФБ; ECC – диаграмма ECC; $Z_B^0 = (Z_{VI}^0, Z_{VO}^0, Z_{VV}^0)$ – набор функций начальных значений, где Z_{VI}^0, Z_{VO}^0 и Z_{VV}^0 – функции начальных значений входных, выходных и внутренних переменных.

Интерфейс ФБ определяется следующим кортежем:

$$Interface = (EI, EO, VI, VO, IW, OW),$$

где $EI = \{ei_1, ei_2, ..., ei_{N_{EI}}\}$ – конечное непустое упорядоченное множество входных событийных переменных с отношением линейного порядка $\prec$, $Dom(EI) = \{true, false\}$; $EO = \{eo_1, eo_2, ..., eo_{N_{EO}}\}$ – конечное непустое множество выходных событийных переменных, $Dom(EO) = \{true, false\}$; $VI = \{vi_1, vi_2, ..., vi_{N_{VI}}\}$ – конечное множество входных информационных переменных, $Dom(VI) = N$, где N – множество целых чисел; $VO = \{vo_1, vo_2, ..., vo_{N_{VO}}\}$ – конечное множество выходных информационных переменных, $Dom(VO) = N$; $IW \subseteq EI \times VI$ – множество *WITH*-связей для входов; $OW \subseteq EO \times VO$ – множество *WITH*-связей для выходов.

Для корректности задания интерфейса должны выполняться условия:

$$VI \setminus Pr_2\, IW = \varnothing \text{ и } VO \setminus Pr_2\, OW = \varnothing.$$

Иными словами, каждый информационный вход или выход должен быть инцидентен какой-либо *WITH*-связи. Имеет место $VI \cap VO \cap VV = \varnothing$. Обозначим $V = VI \cup VO \cup VV$ – множество всех переменных базисного ФБ, а $VVO = VO \cup VV$ – множество тех переменных из *VV*, которые могут быть изменены.

Диаграмма *ECC* может быть определена следующим образом:

$$ECC = (DomQ, ECTranOrd, fGuardCond, fECAction, q_0),$$

где $DomQ = \{q_0,q_1,\ldots,q_m\}$ – конечное непустое множество состояний диаграммы *ECC*; $ECTranOrd = (ECTran, \prec)$ – упорядоченное множество *EC*-переходов с отношением линейного порядка $\prec$. Данный порядок фактически определяется местоположением соответствующего описания *EC*-перехода в XML-файле; $ECTran \subseteq DomQ \times (EI \cup \{\varepsilon\}) \times DomQ$ – базовое множество *EC*-переходов диаграммы *ECC*; кортеж $(q_i,ei_k,q_j) \in ECTran$ означает, что существует *EC*-переход из *EC*-состояния q_i в *EC*-состояние q_j, помеченный входным сигналом ei_k. В случае $(q_i,\varepsilon,q_j) \in ECTran$ *EC*-переходу не назначен никакой триггерный сигнал (случай так называемого нетриггерного *EC*-перехода); $fGuardCond: ECTRan \rightarrow [[Z_V] \rightarrow \{true,false\}]$ – функция, назначающая переходам сторожевые условия; $fECAction: DomQ \rightarrow ((Alg \cup \{\varepsilon\}) \times (EO \cup \{\varepsilon\}))^*$ – функция, назначающая *EC*-сос-тояниям списки *EC*-акций. Здесь под $(A)^*$ понимается множество произвольных последовательностей (любой длины), составленных из элементов множества *A*. Данная нотация заимствована из теории формальных языков и грамматик [2]. Каждая *EC*-акция может включать алгоритм и выходной сигнал. Символом ε отмечено отсутствие соответствующего компонента; $q_0 \in DomQ$ – начальное *EC*-состояние.

Следует отметить, что выше приведено определение нормализованной формы диаграммы *ECC*, в которой не допускается использование двух различных *EC*-переходов между одними и теми же *EC*-состояниями, помеченными одним и тем же входным сигналом или вообще не помеченных входными сигналами. Данных ситуаций можно избежать, используя технику рефакторинга, описанную в работе [246].

Семантическая часть описания МБФБ определяется двойкой:

$$Sem_B = (VRT_B, T_B),$$

где VRT_B – набор переменных времени выполнения; T_B – набор функций переходов МБФБ.

Набор переменных времени выполнения определяется кортежем

$$VRT_B = (VIB, VOB, Q, S, NA, NI, \omega, \alpha, \beta),$$

$VIB = \{vib_1, vib_2, ..., vib_{N_{VI}}\}$ – множество внешних буферов, связанных с входными переменными, $|VIB|=|VI|$, $VIB \leftrightarrow VI$, $Dom(VIB) = N$; $VOB = \{vob_1, vob_2, ..., vob_{N_{VO}}\}$ – множество внешних буферов, связанных с выходными переменными, $|VOB|=|VO|$, $VOB \leftrightarrow VO$, $Dom(VOB) = N$; Q – переменная текущего *EC*-состояния, $Dom(Q) = DomQ$; S – переменная текущего состояния *OSM*-машины, $Dom(S) = \{s_0, s_1, s_2\}$; NA – указатель (счетчик) текущей *EC*-акции, $Dom(NA) = \{0,1,..., \max\limits_{q_i \in DomQ} N_Q^{q_i}\}$, где $N_Q^{q_i}$ – число *EC*-акций в состоянии q_i. Если $NA = 0$, то считается, что ни одна из *EC*-акций не находится в стадии исполнения; NI – указатель (счетчик) текущего шага алгоритма, $Dom(NI) = \{0,1,..., \max\limits_{q_i \in DomQ, j \in 1, N_Q^{q_i}} N_{NI}^{q_i,j}\}$, где $N_{NI}^{q_i,j}$ – число шагов в алгоритме j-й EC-акции в состоянии q_i. Если $NI = 0$, то считается, что алгоритм закончил свое выполнение; ω – признак окончания передач сигналов в объемлющем ФБ, $Dom(\omega) = \{true, false\}$; α – признак запуска модуля диспетчером, $Dom(\alpha) = \{true, false\}$; β – признак окончания работы модуля, $Dom(\beta) = \{true, false\}$.

Набор функций переходов МБФБ определяется как

$$T_B = (t_{EI}, t_{EO}, t_{VI}, t_{VO}, t_{VV}, t_{VOB}, t_Q, t_S, t_{NA}, t_{NI}, t_\alpha, t_\beta),$$

где t_{EI}: $[Z_Q] \times [Z_S] \times [Z_{EI}] \times [Z_V] \rightarrow [Z_{EI}]$ – функция сброса входных событийных переменных;

t_{EO}: $[Z_Q] \times [Z_S] \times [Z_{NA}] \times [Z_{NI}] \rightarrow [Z_{EO}]$ – функция установки выходных событийных переменных;

t_{VI}: $[Z_\alpha] \times [Z_{EI}] \times [Z_S] \times [Z_{VIB}] \rightarrow [Z_{VI}]$ – функция изменения входных переменных (в результате съема данных);

$t_{VO}: [Z_V] \times [Z_S] \times [Z_Q] \times [Z_{NA}] \times [Z_{NI}] \rightarrow [Z_{VO}]$ – функция изменения выходных переменных (в результате выполнения алгоритма);

$t_{VV}: [Z_V] \times [Z_S] \times [Z_Q] \times [Z_{NA}] \times [Z_{NI}] \rightarrow [Z_{VV}]$ – функция изменения внутренних переменных (в результате выполнения алгоритма);

t_{VOB}: $[Z_Q] \times [Z_S] \times [Z_{VO}] \times [Z_{NA}] \times [Z_{NI}] \rightarrow [Z_{VOB}]$ – функция изменения выходных буферов (в результате выдачи данных);

$t_Q: [Z_Q] \times [Z_S] \times [Z_{EI}] \times [Z_V] \rightarrow [Z_Q]$ – функция переходов *EC*-состояний;

$t_S: [Z_\alpha] \times [Z_Q] \times [Z_S] \times [Z_{EI}] \times [Z_V] \times [Z_{NA}] \rightarrow [Z_S]$ – функция переходов *OSM*-состояний;

$t_{NA}:[Z_{NA}]\times[Z_Q]\times[Z_S]\times[Z_{NI}]\to[Z_{NA}]$ – функция изменения указателя *EC*-акций;

$t_{NI}:[Z_{NI}]\times[Z_S]\times[Z_Q]\times[Z_{NA}]\times[Z_V]\to[Z_{NI}]$ – функция изменения указателя инструкций в алгоритме;

$t_\alpha:[Z_\alpha]\times[Z_\omega]\times[Z_S]\times[Z_{EI}]\times[Z_V]\to[Z_\alpha]$– функция сброса признака запуска модуля ФБ;

$t_\beta:[Z_\alpha]\times[Z_\omega]\times[Z_S]\times[Z_{EI}]\times[Z_V]\to[Z_\beta]$ – функция установки признака окончания выполнения ФБ.

Для удобства дальнейшего использования представим функции значений наборов переменных в виде объединения функций значений отдельных переменных:

$$Z_{EI}=\bigcup_{i=1}^{N_{EI}} Z_{ei_i};\ Z_{EO}=\bigcup_{i=1}^{N_{EO}} Z_{eo_i};\ Z_{VI}=\bigcup_{i=1}^{N_{VI}} Z_{vi_i};$$

$$Z_{VO}=\bigcup_{i=1}^{N_{VO}} Z_{vo_i};\ Z_{VV}=\bigcup_{i=1}^{N_{VV}} Z_{vv_i};\ Z_{VOB}=\bigcup_{i=1}^{N_{VOB}} Z_{vob_i}.$$

3.7.3. Определение динамики модели

Ниже представлена динамика функционирования МБФБ в виде правил изменения функций значений переменных.

Определим условие выбора входного сигнала для дальнейшей его обработки ФБ. Назовем этот сигнал *активным*. Как было сказано в первой главе, стандарт IEC 61499 не определяет процедуру выбора активного сигнала. Там же (в первой главе) были предложены несколько стратегий управления входными сигналами, включающими выбор и обработку.

Приоритетное правило выбора входного сигнала $ei_k\in EI$ может быть выражено следующим условием:

$$selectEI_k \triangleq Z_{EI}(ei_k)\wedge \bigwedge_{ei_k\prec ei_j} \overline{Z_{EI}(ei_j)}.$$

В соответствии с данным условием событийный входной сигнал считается активным, если нет сигналов на более приоритетных событийных входах. Недостатком правила приоритетного выбора является то, что может быть выбран входной сигнал, который не является ожидаемым в текущем *EC*-состоянии и поэтому никаких действий в системе не производит. И в то же время могут быть удалены ожидаемые, но менее приоритетные сигналы. Предлагается

более точное правило выбора входного сигнала с учетом ожидаемого сигнала:

$$selectEI_k \triangleq$$

$$\triangleq Z_{EI}(ei_k) \wedge \bigvee_{q_i \in Q^{eik}} (Z_Q(Q) = q_i) \wedge \bigwedge_{ei_k \prec ei_j} \overline{Z_{EI}(ei_j) \wedge \bigvee_{q_m \in Q^{ei\,j}} (Z_Q(Q) = q_m)},$$

где $Q^{ei} \subseteq DomQ$ – множество *EC*-состояний, из которых выходят *EC*-переходы, помеченные сигналом *ei*.

В соответствии с приведенным условием входной сигнал считается активным, если данным сигналом помечен *EC*-переход, выходящий из текущего *EC*-состояния и нет других, более приоритетных сигналов, которые также помечали бы *EC*-переходы, выходяшие из текущего *EC*-состояния. Однако и в данном случае нет гарантии того, что активный сигнал произведет какие-либо действия, поскольку сторожевое условие соответствующего *EC*-перехода может оказаться ложным. Кроме того, в соответствии с данным условием может оказаться невыбранным ни один из входных сигналов.

Более точное правило выбора можно было бы разработать на основе понятия *действующего* активного входного сигнала, т.е. такого сигнала, который однозначно инициирует *EC*-переходы в ФБ. Однако это требует предварительной оценки входных сигналов с соответствующим *пробным съемом* данных. Следует отметить, что данная процедура не предусмотрена стандартом: согласно *OSM* сначала производится выбор сигнала и съем данных, а только потом оценка *EC*-переходов.

Функция t_{EI} определяется принятой дисциплиной сброса входных сигналов, варианты дисциплин были рассмотрены в главе 1. Для дисциплины *DelEI*3, при которой удаляется только тот сигнал, который обрабатывается, правила изменения функции Z_{EI} будут следующими:

$$\{p_{EI}^{B,1}[k] : Z_S(S) = s_1 \wedge selectEI_k \Rightarrow Z_{EI}(ei_k) \leftarrow false \mid ei_k \in EI\}.$$

При использовании дисциплины *DelEI*1, к которой больше склоняется стандарт, в дополнение к вышеприведенному правилу необходимо добавить правила, сбрасывающие все невыбранные входные сигналы:

$$\{p_{EI}^{B,2}[m] : Z_\alpha(\alpha) \wedge Z_S(S) = s_0 \wedge \overline{selectEI_m} \Rightarrow Z_{EI}(ei_m) \leftarrow false \mid ei_m \in EI\}.$$

В стандарте определено, что съем данных осуществляется при переходе *OSM*-машины из состояния *s*0 в *s*1. Предполагается, что

производится съем значений только тех входных переменных, которые связаны с активным событийным входом (с использованием WITH-ассоциаций). Правила съема входных данных в этом случае могут быть выражены следующим образом:

$$\{p_{VI}^{B,1}[m]: Z_\alpha(\alpha) \wedge Z_S(S) =$$
$$= s_0 \wedge \bigvee_{(ei_k, vi_m) \in IW} selectEI_k \Rightarrow Z_{VI}(vi_m) \leftarrow Z_{VIB}(vib_m) \mid vi_m \in VI\}.$$

Далее рассмотрим функционирование машин *OSM* и *ECC*, играющих ключевую роль в определении базисного ФБ. Предварительно для удобства введем ряд предикатов и условий, используемых в правилах модификации функций:

GuardCond: $ECTran \times [Z_V] \rightarrow \{true, false\}$ – предикат, определяющий сторожевые условия *EC*-переходов;

TranCond: $ECTran \times [Z_{EI}] \times [Z_V] \rightarrow \{true, false\}$ – предикат, определяющий условия *EC*-переходов в целом.

Для триггерного *EC*-перехода $ect = (q_i, ei_k, q_j) \in ECTran$, где $q_i, q_j \in DomQ$, $ei_k \in EI$ имеет место следующее определение:

$$TranCond(q_i, ei_k, q_j, Z_{EI}, Z_V) \triangleq Z_{EI}(ei_k) \wedge GuardCond(q_i, ei_k, q_j, Z_V),$$

где Z_{EI} и Z_V – функции текущих значений переменных из множеств *EI* и $V=VI \cup VO \cup VV$. В соответствии с этим выражением условие *EC*-перехода $ect = (q_i, ei_k, q_j) \in ECTran$ является истинным, если имеется сигнал на событийном входе ei_k и сторожевое условие *EC*-перехода истинно.

Для нетриггерного перехода $ect = (q_i, \varepsilon, q_j) \in ECTran$ условие перехода определяется как

$$TranCond(q_i, \varepsilon, q_j, Z_{EI}, Z_V) \triangleq GuardCond(q_i, \varepsilon, q_j, Z_V).$$

В данном случае истинность условия перехода зависит только от сторожевого условия.

Введем условие разрешенности *EC*-переходов. Для триггерного *EC*-перехода $ect = (q_i, ei_k, q_j) \in ECTran$ данное условие имеет вид

$$EnabledECTran(q_i, ei_k, q_j, Z_{EI}, Z_V) \triangleq$$
$$\triangleq selectEI_k \wedge GuardCond(q_i, ei_k, q_j, Z_V).$$

Для нетриггерного перехода $ect = (q_i, \varepsilon, q_j) \in ECTran$ условие разрешенности совпадает со сторожевым условием

$$EnabledECTran(q_i, \varepsilon, q_j, Z_{EI}, Z_V) \triangleq GuardCond(q_i, ei_k, q_j, Z_V).$$

*У*словия наличия разрешенных *EC*-переходов в текущем *EC*-состоянии *ExistsEnabledECTran*, а также условие их отсутствия *AbsentEnabledECTran* представлены ниже:

$$ExistsEnabledECTran \triangleq$$

$$\triangleq \bigvee_{q_i \in DomQ} (Z_Q(Q) = q_i \wedge \bigvee_{(q_i,x,q_j) \in ECTran} EnabledECTran(q_i, x, q_j, Z_{EI}, Z_V)$$

$$AbsentsEnabledECTran \triangleq$$

$$\triangleq \bigvee_{q_i \in DomQ} (Z_Q(Q) = q_i \wedge \bigwedge_{(q_i,x,q_j) \in ECTran} \overline{EnabledECTran(q_i, x, q_j, Z_{EI}, Z_V)}.$$

Функция переходов T_S реализуется в виде четырех (бесприоритетных) правил, каждое из которых соответствует одному переходу *OSM*-машины [4]:

$$p_S^{B,1} : Z_\alpha(\alpha) \wedge Z_S(S) = s_0 \wedge \bigvee_{ei_k \in EI} selectEI_k \Rightarrow Z_S(S) \leftarrow s_1;$$

$$p_S^{B,2} : Z_S(S) = s_1 \;\&\; ExistsEnabledECTran \Rightarrow Z_S(S) \leftarrow s_2;$$

$$p_S^{B,3} : Z_S(S) = s_2 \;\&\; Z_{NA}(NA) = 0 \Rightarrow Z_S(S) \leftarrow s_1;$$

$$p_S^{B,4} : Z_S(S) = s_1 \;\&\; AbsentsEnabledECTran \Rightarrow Z_S(S) \leftarrow s_0.$$

Согласно правилу $p_S^{B,1}$ *OSM*-машина стартует, когда есть сигнал запуска ФБ от диспетчера и существует хотя бы один выбранный входной событийный сигнал. В соответствии с простейшими дисциплинами выбора достаточно наличия хотя бы одного сигнала на событийных входах. По правилу $p_S^{B,2}$ *OSM*-машина переходит в состояние s_2 (состояние выполнения *EC*-акций), если она находится в состоянии s_1 и существует хотя бы один разрешенный *EC*-переход из текущего *EC*-состояния. В соответствии с правилом $p_S^{B,3}$ *OSM*-машина возвращается в состояние s_1, если выполнены все *EC*-акции, ассоциированные с текущим *EC*-состоянием. Правило $p_S^{B,4}$ определяет, что *OSM*-машина возвращается в состояние s_0, если в состоянии s_1 нет разрешенных *EC*-переходов.

Правила, определяющие реализационные составляющие функции перехода t_Q, представлены ниже в виде группы правил с приоритетами:

$$< p_Q^{B,1}[i,x,j] : Z_Q(Q) = q_i \wedge Z_S(S) =$$
$$= s_1 \wedge EnabledECTran(q_i, x, q_j, Z_{EI}, Z_V) \Rightarrow$$
$$\Rightarrow Z_Q(Q) \leftarrow q_j \mid (q_i, x, q_j) \in ECTranOrd >.$$

В данном приоритетном наборе столько правил, сколько в диаграмме *ECC* переходов. Упорядочивание этих правил в наборе производится согласно отношению порядка, используемому в упорядоченном множестве *ECTranOrd*. В соответствии с правилом из данного набора *EC*-переход может сработать, если условие разрешенности перехода истинно и *OSM*-машина находится в состоянии s_1. Назовем выбранный для срабатывания *EC*-переход *активным*. Для реализации при вычислении значений переменной Q может быть удобнее использовать следующее множество правил, не связанных явно с приоритетами:

$$\{p_Q^{B,2}[j]: Z_S(S) =$$

$$= s_1 \wedge \bigvee_{(q_i,x,q_j)\in ECTran} (Z_Q(Q) = q_i \wedge EnabledECTran(q_i, x, q_j, Z_{EI}, Z_V) \wedge$$

$$\wedge \bigwedge_{\substack{(q_i,y,q_m)\in ECTran,\\ (q_i,x,q_j)\prec(q_m,y,q_j)}} \overline{EnabledECTran(q_i, y, q_m, Z_{EI}, Z_V)}) \Rightarrow$$

$$\Rightarrow Z_Q(Q) \leftarrow q_j \mid q_j \in DomQ\}.$$

В данном случае знак $\prec$ определяет приоритетность *EC*-переходов. Стоящий слева от этого знака переход менее приоритетный, чем стоящий справа.

Правила, определяющие t_{NA} – функцию изменения счетчика *EC*-акций, приводятся ниже:

$$p_{NA}{}^{B,1}: Z_S(S) = s_1 \Rightarrow Z_{NA}(NA) \leftarrow 1;$$

$$\{p_{NA}{}^{B,2}[i]: Z_S(S) = s_2 \wedge Z_{NI}(NI) = 0 \wedge Z_Q(Q) = q_i \wedge Z_{NA}(NA) < N_A{}^{qi} \Rightarrow$$

$$\Rightarrow Z_{NA}(NA) \leftarrow Z_{NA}(NA)+1 \mid q_i \in Q\};$$

$$\{p_{NA}{}^{B,3}[i]: Z_S(S) = s_2 \wedge Z_{NI}(NI) = 0 \wedge Z_Q(Q) = q_i \wedge Z_{NA}(NA) = N_A{}^{qi} \Rightarrow$$

$$\Rightarrow Z_{NA}(NA) \leftarrow 0 \mid q_i \in Q\},$$

где $N_A{}^{qi}$ – число *EC*-акций в *EC*-состоянии q_i.

Как видно из вышеприведенных правил, счетчик *EC*-акций *NA* устанавливается в единицу в состоянии s_1 *OSM*-машины. Увеличение счетчика *EC*-акций производится в случае завершения работы алгоритма ($NI = 0$) при условии, что выполняемая *EC*-акция не является последней. Если же эта *EC*-акция является последней, то счетчик *EC*-акций сбрасывается в ноль.

Для дальнейшего изложения введем предикат, определяющий условие изменения указателя *NI* в алгоритмах при условном переходе:

$$CondNI: Dom(Q) \times Dom(NA) \times Dom(NI) \times [Z_V] \rightarrow \{true, false\},$$

и функцию, позволяющую вычислить номер следующей выполняемой инструкции в алгоритмах, если условие перехода будет выполнено:

$$nextSt\colon Dom(Q) \times Dom(NA) \times Dom(NI) \times [Z_V] \to Dom(NI).$$

Данные предикат и функция являются априори заданными и, по сути дела, определяются самими алгоритмами ФБ. По окончании выполнения алгоритма значение счетчика шагов инструкций *NI* сбрасывается в ноль. В случае использования только линейных алгоритмов определение условий перехода *CondNI* и функции *nextSt* не требуется. В этом случае, если не достигнут конец алгоритма, счетчик инструкций *NI* просто увеличивается на единицу.

Правила для реализации t_{NI}-функции изменения счетчика инструкций в алгоритмах представлены ниже:

$$p_{NI}^{B,1} : Z_S(S) = s_1 \Rightarrow Z_{NI}(NI) \leftarrow 1;$$

$$\{p_{NI}^{B,2}[i,j,k] : Z_S(S) = s_2 \wedge Z_Q(Q) =$$

$$= q_i \wedge Z_{NA}(NA) = j \wedge Z_{NI}(NI) = k \wedge$$

$$CondNI(q_i, j, k, Z_V) \Rightarrow Z_{NI}(NI) \leftarrow$$

$$\leftarrow nextSt(q_i, j, k, Z_V) \mid q_i \in DomQ, j \in \overline{1, N_A^{q_i}}, k \in \overline{1, N_I^{q_i, j}}\};$$

$$p_{NI}^{B,3} : Z_S(S) = s_2 \wedge \bigvee_{q_i \in DomQ} \overline{\bigvee_{j \in 1, N_A^{q_i}}} \overline{\bigvee_{k \in 1, N_I^{q_i, j}}} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) =$$

$$= j \wedge Z_{NI}(NI) = k \wedge \overline{CondNI(q_i, j, k, Z_V)} \Rightarrow Z_{NI}(NI) \leftarrow Z_{NI}(NI) + 1,$$

где $N_A^{q_i}$ – число *EC*-акций в состоянии q_i; $N_I^{q_i, j}$ – число инструкций в *j*-й *EC*-акции состояния q_i.

Как видно из данных правил, в состоянии s_1 *OSM*-машины счетчик числа инструкций устанавливается в единицу. В состоянии s_2 (на этапе выполнения алгоритмов) значения счетчика изменяются в зависимости от текущего значения и функции его изменения. Если в шаге алгоритма используется условный переход и при оценке условия перехода *CondNI* получено значение «Истина», то значение счетчика *NI* изменяется согласно функции *NextSt*. Иначе счетчик *NI* просто увеличивается на единицу.

В случае использования линейных алгоритмов можно использовать следующий упрощенный набор правил $< p_{NI}^{B,4}, p_{NI}^{B,5}, p_{NI}^{B,6}, p_{NI}^{B,7} >$:

$$p_{NI}^{B,4} : Z_S(S) = s_1 \Rightarrow Z_{NI}(NI) \leftarrow 1;$$

$$p_{NI}^{B,5}: Z_S(S) = s_2 \wedge \bigvee_{q_i \in DomQ,\, j \in \overline{1, N_A^{q_i}}} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) =$$

$$= j \wedge Z_{NI}(NI) < N_I^{q_i, j}) \Rightarrow Z_{NI}(NI) \leftarrow Z_{NI}(NI) + 1;$$

$$p_{NI}^{B,6}: Z_S(S) = s_2 \wedge \bigvee_{q_i \in DomQ,\, j \in \overline{1, N_A^{q_i}}} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) =$$

$$= j \wedge Z_{NI}(NI) = N_I^{q_i, j}) \Rightarrow Z_{NI}(NI) \leftarrow 0;$$

$$p_{NI}^{B,7}: Z_{NI}(NI) = 0 \Rightarrow Z_{NI}(NI) \leftarrow 1.$$

Последнее правило определяет, что нулевое значение переменной *NI* недолговечно и в следующем такте сразу переключается на единицу. Нулевое значение *NI* необходимо: оно используется, например, в определении завершения выполнения *EC*-акции и выдачи выходных сигналов.

Определим функцию t_{VO} в виде правил изменения функции Z_{VO}. Для этого введем предикаты, определяющие условия изменения переменных $vo_m \in VO\ (m = \overline{1, N_{VO}})$ в алгоритмах:

$$CondVO_m: Dom(Q) \times Dom(NA) \times Dom(NI) \times [Z_V] \rightarrow \{true, false\},$$

а также функции изменения значений переменных $vo_m \in VO$ $(m = \overline{1, N_{VO}})$ с учетом всех алгоритмов, выполняющихся во всевозможных *EC*-состояниях и *EC*-акциях:

$$NewValVO_m: Dom(Q) \times Dom(NA) \times Dom(NI) \times [Z_V] \rightarrow Dom(V).$$

Правила для реализации функции t_{VO} представлены ниже:

$$\{p_{VO}^{B,1}[i,j,k,m]: Z_S(S) = s_2\ \&\ Z_Q(Q) = q_i \wedge Z_{NA}(NA) = j \wedge Z_{NI}(NI) = k \wedge$$
$$CondVO_m(q_i, j, k, Z_V) \Rightarrow Z_{VO}(vo_m) \leftarrow NewValVO_m(q_i, j, k, Z_V)$$
$$\mid vo_m \in VO, q_i \in Q^{vo_m}, j \in NA^{q_i, vo_m}, k \in NI^{q_i, j, vo_m}\},$$

где $Q^{vo_m} \subseteq DomQ$ – множество *EC*-состояний, в которых может производиться изменение переменной vo_m; NA^{q_i, vo_m} – множество номеров *EC*-акций в *EC*-состоянии q_i, в которых может производиться изменение переменной vo_m; NI^{q_i, j, vo_m} – множество номеров шагов в *j*-й *EC*-акции в *EC*-состоянии q_i, в которых может производиться изменение переменной vo_m.

Аналогичные правила могут быть составлены для переменных из множества *VV* (данные правила не приводятся).

Определим условия выдачи выходных сигналов eo_k ($k = \overline{1, N_{EO}}$) следующим образом:

$$putoutEO_k \triangleq Z_S(S) = s_2 \wedge Z_{NI}(NI) = 0 \wedge \bigvee_{q_i \in Q^{eo_k}} \bigvee_{j \in NA^{q_i, eo_k}} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) = j),$$

где $Q^{eo_k} \subseteq DomQ$ – множество *EC*-состояний, хотя бы в одной *EC*-акции которых выдается выходной сигнал eo_k; NA^{q_i, eo_k} – множество номеров *EC*-акций *EC*-состояния q_i, в которых выдается выходной сигнал eo_k.

Правила для выдачи выходных сигналов с использованием условий $putoutEO_k$ могут быть выражены так:

$$\{p_{EO}^{B,1}[k]: putoutEO_k \Rightarrow Z_{EO}(eo_k) \leftarrow true \mid eo_k \in EO\}.$$

При выдаче выходного сигнала производится сопутствующая ему передача выходных данных из ФБ в выходные буфера, соответствующие правила приведены ниже:

$$\{p_{VOB}^{B,1}[m]: \bigvee_{(eo_k, vo_m) \in OW} putoutEO_k \Rightarrow Z_{VOB}(vob_m) \leftarrow Z_{VO}(vo_m) \mid vo_m \in VO\}.$$

Сигнал об окончании выполнения базисного ФБ должен формироваться в двух случаях: 1) в случае, когда при приходе стартового сигнала α на ФБ, находящийся в состоянии «Свободен», оказывается, что на его событийных входах нет действующих сигналов (так называемый случай «пустого» запуска); 2) при переходе *OSM*-машины из состояния s_1 в состояние s_0 (случай нормальной отработки ФБ).

Формирование сигнала окончания работы базисного ФБ (с использованием функции t_β) может быть определено следующими правилами:

$$p_\beta^{1,B}: Z_\alpha(\alpha) \wedge Z_\omega(\omega) \wedge Z_S(S) = s_0 \wedge \bigwedge_{ei_k \in EI} \overline{selectEI_k} \Rightarrow Z_\beta(\beta) \leftarrow true;$$

$$p_\beta^{2,B}: Z_S(S) = s_1 \wedge AbsentsEnabledECTran \Rightarrow Z_\beta(\beta) \leftarrow true.$$

Следует отметить, что сброс переменной β в «ложь» производится в диспетчере.

Одновременно с установкой признака β сбрасывается признак запуска α. Правила для изменения α аналогичны правилам для изменения β, но ниже они представлены в виде одного правила:

$$p_{\alpha}^{1,B}:(Z_{\alpha}(\alpha)\wedge Z_{\omega}(\omega)\wedge Z_S(S)=s_0\wedge \bigwedge_{ei_k\in EI}\overline{selectEI_k}\vee$$

$$\vee\,(Z_S(S)=s_1\wedge AbsentsEnabledECTran)\Rightarrow Z_{\alpha}(\alpha)\leftarrow false.$$

3.8. Модель составного функционального блока для циклической модели выполнения

3.8.1. Определение схемы модели

Модуль (или модель) составного ФБ (МСФБ) может быть представлен формально следующим образом:

$$M_C^C=(Synt_C,Sem_C^C),$$

где $Synt_C$ – синтаксическая часть описания (на уровне абстрактного синтаксиса); Sem_C^C – семантическая часть описания.

Синтаксическая часть описания МСФБ определяется четверкой:

$$Synt_C = (Interface, FB, EvConn, DataConn),$$

где *Interface* – интерфейс составного ФБ (аналогичен интерфейсу базисного ФБ, определенного выше); $FB=\{fb_1,fb_2,...,fb_{N_{FB}}\}$ – множество компонентных ФБ, входящих в составной ФБ, fb_i = ($Interface^i$, fbt^i), $i\in[1,N_{FB}]$, где $Interface^i$ = (EI^i, EO^i, VI^i, VO^i) – интерфейс компонентного ФБ. В данный интерфейс входят: EI^i и EO^i – множества входных и выходных событийных переменных; VI^i и VO^i – множества входных и выходных переменных *i*-го компонентного ФБ соответственно; fbt^i – тип *i*-го компонентного ФБ; $EvConn\subseteq$ $\subseteq(EI\cup\bigcup_{i=1}^{N_{FB}}EO^i)\times(EO\cup\bigcup_{i=1}^{N_{FB}}EI^i)$ – множество событийных связей; $DataConn\subseteq(VI\cup\bigcup_{i=1}^{N_{FB}}VO^i)\times\bigcup_{i=1}^{N_{FB}}VI^i\cup\bigcup_{i=1}^{N_{FB}}VO^i\times VO$ – множество информационных связей. Причем для информационных связей должно выполняться условие

$$\forall(p,t),(q,u)\in EvConn[(t=u)\rightarrow(p=q)].$$

Иными словами, к одному информационному входу нельзя подключить более одного информационного выхода. При использо-

вании событийных связей подобных топологических ограничений на структуру не накладывается, поскольку подразумевается неявное использование блоков *E_SPLIT* и *E_MERGE* для расщепления и слияния событий соответственно.

Семантическая часть описания МСФБ определяется тройкой:

$$Sem_C^C = (VRT_C^C, T_C^C, D_C^C),$$

где VRT_C^C – набор переменных времени выполнения составного ФБ; T_C^C – набор функций переходов МСФБ; D_C^C – диспетчер, определяющий порядок выполнения компонентных ФБ внутри родительского составного ФБ в соответствии с циклической моделью выполнения.

Набор переменных времени выполнения определяется кортежем

$$VRT_C^C = (VIB, VOB, FBD^C, \omega, \alpha, \beta),$$

где *VIB*, *VOB*, α, β имеют тот же смысл, что и в МБФБ; $\omega \triangleq \bigwedge_{eo_j^k \in \bigcup_i EO_i} \overline{Z_{EO}(eo_j^k)} \wedge \bigwedge_{ei_j \in EI} \overline{Z_{EI}(ei_j)}$ – условие окончания передач сигналов в составном ФБ; $FBD^C = \{fbd_1, fbd_2, ..., fbd_{N_{FB}}\}$ – множество дополнительных (семантических) описаний компонентных ФБ, входящих в состав составного ФБ, $FB \leftrightarrow FBD^C$, $fbd_i = (\alpha_i, \beta_i)$, где α_i – переменная запуска *i*-го компонентного ФБ; β_i – переменная окончания работы *i*-го компонентного ФБ.

Набор функций переходов МСФБ определяется как

$$T_C = (t_{EI}, (t_{EI^i})_{i=\overline{1,N_{FB}}}, (t_{EO^i})_{i=\overline{1,N_{FB}}}, t_{EO}, t_{VI}, t_{VOB}),$$

где $t_{EI} : [Z_{EI}] \times [Z_\alpha] \to [Z_{EI}]$ – функция сброса входных событийных переменных модуля (в результате передачи сигналов); $t_{EI^i} : [Z_\alpha] \times \times [\bigcup_{i=1}^{N_{FB}} Z_{EO^i}] \times [Z_{EI}] \to [Z_{EI_i}]$ – функция установки входных событийных переменных *i*-го компонентного ФБ; $t_{EO^i} : [Z_{EO^i}] \to [Z_{EO^i}]$ – функция сброса выходных событийных переменных *i*-го компонентного ФБ; $t_{EO} : [\bigcup_{i=1}^{N_{FB}} Z_{EO^i}] \times [Z_{EI}] \to [Z_{EO}]$ – функция установки выходных событийных переменных модуля (в результате передачи сигналов на его выходы); t_{VI}: $[Z_\alpha] \times [Z_{VIB}] \times [Z_{EI}] \to [Z_{VI}]$ –

функция изменения входных переменных (в результате съема данных); $t_{VOB}:[\bigcup_{i=1}^{N_{FB}} Z_{VO^i}]\times[Z_{EI}]\times[\bigcup_{i=1}^{N_{FB}} Z_{EO^i}]\to[Z_{VOB}]$ – функция изменения выходных буферов (в результате выдачи данных).

3.8.2. Определение динамики модели

Ниже кратко представлены правила функционирования составного ФБ в рамках некоторой типичной последовательной модели выполнения, в которой выбор активного ФБ определяется как функция признаков завершения выполения ФБ. К данному типу моделей выполнения относится, например, циклическая модель. При выполнении МСФБ будем считать, что все действия по передаче сигналов и данных между компонентными ФБ между собой, а также внешней средой производятся мгновенно, точнее – за один логический такт времени. В том числе одновременно производится съем данных в отношении тех событийных входов, на которых стоят сигналы. Это так называемый «*синхронный съем данных*». Таким образом, в отличие от базисного ФБ в данном случае обрабатываются все входные сигналы. Тем не менее это не отвергает возможность использования других дисциплин выбора и обработки входных сигналов, в том числе приоритетных. Передача сигналов и данных на выходы ФБ в данном случае не требует наличия сигнала запуска α, т.е. здесь при передаче данных используется принцип «*горячей картошки*» [134], в соответствии с которым сигналы передаются дальше сразу, как только они появляются. Следует, однако, заметить, что эти опции являются вариативной частью и могут варьироваться от одной модели выполнения к другой. В то же время для передачи сигналов и данных с входов ФБ необходим сигнал запуска ФБ.

Функция передачи сигналов на входы *j*-го компонентного ФБ (функция t_{EI^j}) может быть определена следующим образом:

$$\{p^{C,C,1}_{EI^j}[k]: Z_\alpha(\alpha)\wedge(\bigvee_{\substack{ei_m\in EI,\\ (ei_m,ei^j_k)\in EvConn}} Z_{EI}(ei_m)\vee \bigvee_{\substack{eo^x_n\in EO^x,\\ (eo^x_n,ei^i_k)\in EvConn}} Z_{EO^x}(eo^{\tilde{o}}_n))\Rightarrow$$

$$\Rightarrow Z_{EI^j}(ei^j_k)\leftarrow true \,|\, ei^j_k\in EI^j\},\ j=\overline{1,N_{FB}},$$

где ei^j_k – *k*-я входная событийная переменная *j*-го компонентного ФБ.

В соответствии с данным правилом входная событийная переменная компонентного ФБ устанавливается в единицу, если в еди-

ницу установлена хотя бы одна событийная переменная, связанная с данной событийной переменной событийной связью.

Похожим образом может быть определена функция передачи сигналов на выходы МСФБ (функция t_{EO}):

$$\{p_{EO}^{C,C,1}[k]: \bigvee_{\substack{ei_m \in EI, \\ (ei_m, eo_k) \in EvConn}} Z_{EI}(ei_m) \vee \bigvee_{\substack{eo_n^x \in EO^x, \\ (eo_n^x, eo_k) \in EvConn}} Z_{EO^x}(eo_n^x) \Rightarrow$$

$$\Rightarrow Z_{EO}(eo_k) \leftarrow true \mid eo_k \in EO\}.$$

Функция съема данных МСФБ (функция t_{VI}) может быть реализована с помощью следующего семейства правил:

$$\{p_{VI}^{C,C,1}[m]: Z_{\alpha}(\alpha) \wedge \bigvee_{\substack{ei_k \in EI, \\ (ei_k, vi_m) \in IW}} Z_{EI}(ei_k) \Rightarrow Z_{VI}(vi_m) \leftarrow Z_{VIB}(vib_m) \mid vi_m \in VI\}.$$

Функция выдачи данных МСФБ (функция t_{VOB}) может быть задана следующими правилами:

$$\{p_{VOB}^{C,C,1}[k]: \bigvee_{(eo_k, vob_m) \in OW} (\bigvee_{\substack{ei_j \in EI, \\ (ei_j, eo_k) \in EvConn}} Z_{EI}(ei_j) \vee \bigvee_{\substack{eo_n^x \in EO^x, \\ (eo_n^x, eo_k) \in EvConn}} Z_{EO^x}(eo_n^x)) \Rightarrow$$

$$\Rightarrow Z_{VOB}(vob_m) \leftarrow Z_{VO}(repr_{VO}(vob_m)) \mid vob_m \in VOB\}.$$

Здесь в качестве аргумента функции Z_{VO} используется не выходная переменная из множества VO, а ее представитель. Целью данной подмены является минимизация числа переменных без потери корректности, что подробнее объяснено ниже.

Функции сброса источников сигналов могут быть представлены следующими правилами:

$$\{p_{EI}^{C,C,1}[j]: Z_{\alpha}(\alpha) \wedge Z_{EI}(ei_j) \Rightarrow Z_{EI}(ei_j) \leftarrow false \mid ei_j \in EI\};$$

$$\{p_{EO^j}^{C,C,1}[k]: Z_{EO^j}(eo_k^j) \Rightarrow Z_{EO^j}(eo_k^j) \leftarrow false \mid eo_k^j \in EO^j\}, j = \overline{1, N_{FB}}.$$

Представленные выше функции передачи сигналов и данных $(t_{EI_i})_{i=\overline{1,N_{FB}}}$, t_{EO} и t_{VOB} ориентированы на передачу сигналов и данных по принципу «горячей картошки» [131].

3.9. Модель диспетчера для циклической модели выполнения

Диспетчер для циклической модели выполнения в общем виде определяется в виде тройки

$$D_C = (V_D^C, T_D^C, Z_D^{C,0}),$$

где V_D^C – множество переменных диспетчера; T_D^C – множество функций переходов диспетчера; $Z_D^{C,0}$ – множество функций начальных значений переменных диспетчера. Предполагаем, что диспетчер выполняется синхронно с родительским ФБ.

Множество переменных «циклического» диспетчера определяется как

$$V_D^C = (\alpha, \beta, (\alpha_i)_{i=\overline{1,N_{NF}}}, (\beta_i)_{i=\overline{1,N_{FB}}}),$$

где α – входная переменная запуска подсистемы (составного ФБ или субприложения) с верхнего уровня; β – выходная переменная окончания выполнения контролируемой сети ФБ; $(\alpha_i)_{i=\overline{1,N_{NF}}}$ – выходные переменные запуска компонентных ФБ, входящих в контролируемую сеть ФБ; $(\beta_i)_{i=\overline{1,N_{FB}}}$ – входные переменные окончания выполнения компонентных ФБ контролируемой сети ФБ.

Начальные значения всех переменных равны *false*.

Функции переходов циклического диспетчера определяются в виде кортежа

$$T_D^C = (t_\alpha, t_\beta, (t_{\alpha_i})_{i=\overline{1,N_{FB}}}, (t_{\beta_i})_{i=\overline{1,N_{FB}}}),$$

где $t_\alpha : [Z_{\beta_{N_{FB}}}] \to [Z_\alpha]$ – функция сброса переменной запуска подсистемы; $t_\alpha : [Z_{\beta_{N_{FB}}}] \to [Z_\beta]$ – функция установки переменной окончания выполнения подсистемы; $t_{\alpha_1} : [Z_\alpha] \to [Z_{\alpha_1}]$ и $t_{\alpha_i} : [Z_{\beta_{i-1}}] \to$ $\to [Z_{\alpha_i}]$, $i = \overline{2, N_{FB}}$ – функции установки переменных α_i; $t_{\beta_i} : [Z_{\beta_i}] \to$ $\to [Z_{\beta_i}]$, $i = \overline{1, N_{FB}}$ – функции сброса переменных β_i.

Функционирование циклического диспетчера промежуточного уровня определяется следующими правилами:

$$p_{\alpha_1}^{D,C,1} : Z_\alpha(\alpha) \Rightarrow Z_{\alpha_1}(\alpha_1) \leftarrow true;$$

$$\{p_{\alpha_i}^{D,C,2}[i] : Z_{\beta_{i-1}}(\beta_{i-1}) \Rightarrow Z_{\alpha_i}(\alpha_i) \leftarrow true \mid i = \overline{2, N_{FB}}\};$$

$$\{p_{\beta_i}^{D,C,1}[i] : Z_{\beta_i}(\beta_i) \Rightarrow Z_{\beta_i}(\beta_i) \leftarrow false \mid i = \overline{1, N_{FB}}\};$$

$$p_\beta^{D,C,1} : Z_{\beta_{N_{FB}}}(\beta_{N_{FB}}) \Rightarrow Z_\beta(\beta) \leftarrow true;$$

$$p_\alpha^{D,C,1} : Z_{\beta_{N_{FB}}}(\beta_{N_{FB}}) \Rightarrow Z_\alpha(\alpha) \leftarrow false.$$

Следует отметить, что номер i определяет порядок выполнения ФБ, поэтому перед началом моделирования блоки должны быть упорядочены и им должны быть присвоены порядковые номера.

В отличие от диспетчера промежуточного уровня диспетчер верхнего уровня является независимым от других диспетчеров. Как только будет выполнен последний ФБ в списке главного диспетчера, он приступает к выполнению первого блока этого списка. Данный процесс повторяется циклически. Модель главного диспетчера определяется аналогично модели диспетчера промежуточного уровня, но в ней отсутствуют переменные α и β, а также функции их модификации.

Правила функционирования главного диспетчера включают правила $p_{\alpha_i}^{D,C,2}[i]$, $p_{\beta_i}^{D,C,1}[i]$, а также правило

$$p_{\alpha_i}^{D,C,3} : Z_{\beta_{N_{FB}}}(\beta_{N_{FB}}) \Rightarrow Z_{\alpha_1}(\alpha_1) \leftarrow true.$$

3.10. Взаимосвязь модулей функциональных блоков

Рассмотренные выше модули ФБ (МБФБ и МСФБ) являются взаимосвязанными в системе через переменные интерфейсов. Причем одна и та же переменная может использоваться как в родительском, так и в дочернем модуле. Однако локализация переменной должна быть только в одном модуле, за границами модуля эта же переменная будет использоваться в качестве параметра (как представитель). Таким образом, для связи модулей может использоваться механизм передачи параметров.

На рис. 3.11, 3.12 схематично представлены основные переменные, используемые в МБФБ и МСФБ соответственно, а также их взаимосвязи.

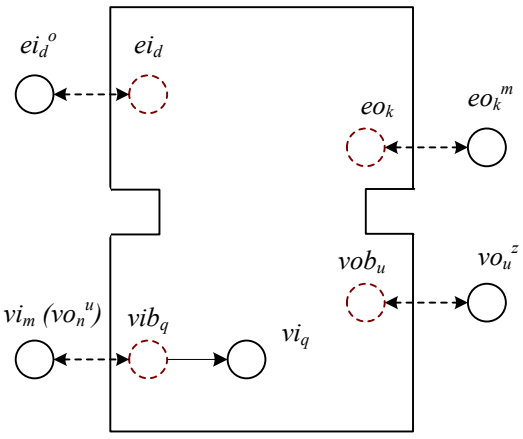


Рис. 3.11. Переменные МБФБ

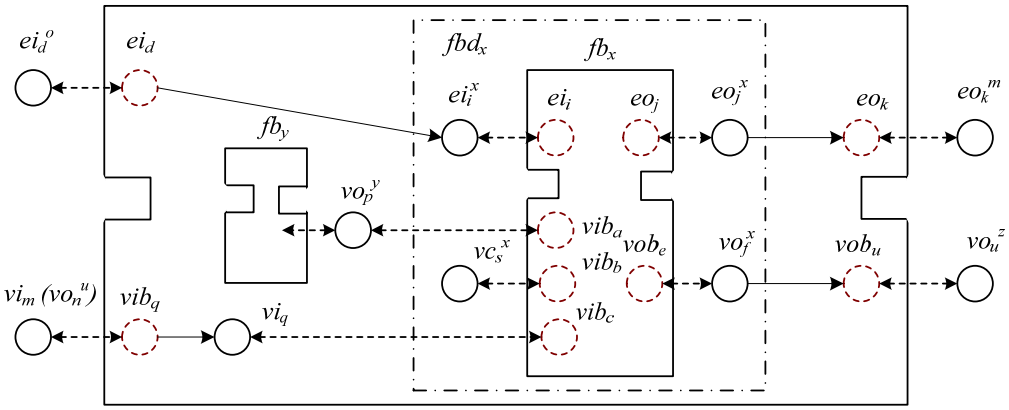


Рис. 3.12. Переменные МСФБ

На приведенных рисунках сплошными окружностями обозначены реальные переменные, а пунктирными – параметры (представители). Взаимосвязи реальных переменных и параметров показаны двунаправленными пунктирными стрелками, а передачи значений переменных – сплошными стрелками. Сплошные стрелки, по сути дела, определяют передачу сигналов и данных в модуль ФБ.

Введем функцию $repr_{VI}$, позволяющую определить представителей входных переменных компонентных ФБ в МСФБ:

$$repr_{VI} : \bigcup_{i=1}^{N_{FB}} VI^i \to VI \cup \bigcup_{i=1}^{N_{FB}} VO^i .$$

Введем функцию $repr_{VO}$, определяющую представителей выходных переменных МСФБ. Поскольку между выходными переменными и выходными буферами существует взаимно однозначное соответствие, то используем при определении функции именно выходные буфера:

$$repr_{VO} : VOB \to \bigcup_{i=1}^{N_{FB}} VO^i .$$

Существуют следующие взаимно однозначные соответствия между переменными i-го компонентного ФБ и соответствующего ему модуля x:

$$EI_i \leftrightarrow EI^x;\ EO_i \leftrightarrow EO^x;\ VO_i \leftrightarrow VOB^x;\ \bigcup_{vi \in VI_i} repr_{VI}(vi) \leftrightarrow VIB^x .$$

В приведенных выше выражениях переменные в левых и правых частях соответствий физически представляют одни и те же переменные.

Соответствие между представителями выходных переменных и выходными буферами $\bigcup_{vob \in VOB} repr_{VO}(vob) \leftrightarrow VOB$ физически выражается в наличии явных передач данных между этими типами переменных.

3.11. Модель составного функционального блока для синхронной модели выполнения

3.11.1. Определение схемы модели

В дальнейшем будем формально определять двухтактную синхронную модель выполнения, рассмотренную выше. МСФБ для двухтактной синхронной модели выполнения определяется как

$$M_C^S = (Synt_C, Sem_C^S),$$

где $Synt_C$ – синтаксическая часть описания (такая же, как и для циклической модели); $Sem_C^S = (VRT_C^S, T_C^S, D_C^S)$ – семантическая часть описания для синхронной модели выполнения, где компоненты набора имеют такой же смысл, что и для набора Sem_C^C, но только в отношении синхронной модели выполнения.

Набор переменных времени выполнения определяется кортежем

$$VRT_C^S = (VIB, VOB, FBD^S, \alpha, \beta, \mu, \psi, \omega),$$

где *VIB* и *VOB* имеют тот же смысл, что и для МСФБ циклической модели; $FBD^S = \{fbd_1, fbd_2, ..., fbd_{N_{FB}}\}$ – множество дополнительных (семантических) описаний компонентных ФБ, входящих в состав составного ФБ. Данное множество делится на два подмножества: $BFBD^S$ – множество описаний базисных компонентных ФБ; $CFBD^S$ – множество описаний составных компонентных ФБ:

$$FBD^S = BFBD^S \cup CFBD^S; \quad BFBD^S \cap CFBD^S = \varnothing.$$

Если $fbd_i \in BFBD^S$, то $fbd_i = (\alpha_i, \beta_i)$, где переменные α_i и β_i имеют такой же смысл, что и для МСФБ в циклической модели.

Если $fbd_j \in CFBD^S$, то $fbd_j = (\alpha_j, \beta_j, \mu_j, \psi_j)$, где α_j (μ_j) – переменная запуска первой (второй) фазы выполнения *j*-го составного компонентного ФБ; β_j (ψ_j) – переменная окончания первой (второй) фазы выполнения *j*-го составного компонентного ФБ; α (μ) – переменная запуска первой (второй) фазы выполнения модуля, $\mathrm{Dom}(\alpha) = \mathrm{Dom}(\mu) = \{true, false\}$; β (ψ) – переменная окончания первой (второй) фазы выполнения модуля, $\mathrm{Dom}(\beta) = \mathrm{Dom}(\psi) = \{true, false\}$; $\omega \triangleq \bigwedge_{eo_j^k \in \bigcup_i EO^i} \overline{Z_{EO}(eo_j^k)} \wedge \bigwedge_{ei_j \in EI} \overline{Z_{EI}(ei_j)}$ – условие окончания передач сигналов в составном ФБ.

Набор функций переходов по сравнению с T_C^C остается тем же самым по структуре: $T_C^S = (t_{EI}, (t_{EI^i})_{i=\overline{1,N_{FB}}}, (t_{EO^i})_{i=\overline{1,N_{FB}}}, t_{EO}, t_{VI}, t_{VOB})$, но сами функции модифицируются. Схема функции изменения выходных событийных переменных t_{EO} принимает вид: $t_{EO}:[Z_{EI}]\times \times[\bigcup_{i=1}^{N_{FB}} Z_{EO^i}]\times[Z_\mu]\times[\bigcup_{k=1}^{N_{CFB}} Z_{\psi_k}]\to[Z_{EO}]$. Эта функция устанавливает выходные событийные переменные модуля в результате выполнения второй фазы. Функция сброса выходных событийных переменных *i*-го компонентного ФБ модифицируется путем учета в ее описании условий завершения второй фазы выполнения ФБ: $t_{EO^i}:[Z_{EO^i}]\times[\bigcup_{i=1}^{N_{CFB}} Z_{\psi_i}]\times[Z_\mu]\to[Z_{EO^i}]$. Функции t_{EI^i} и t_{EI} остаются такими же, как и в наборе T_C^C для МСФБ для циклической модели выполнения.

3.11.2. Определение динамики модели

Ниже приводятся правила изменения функций из набора T_C^S при использовании диспетчера промежуточного уровня.

Правила изменения входных событийных переменных *j*-го компонентного ФБ:

а) для случая, когда в составном ФБ имеется хотя бы один составной компонентный ФБ:

$$\{p_{EI^j}^{C,S,1}[k]: Z_\alpha(\alpha)\wedge \bigvee_{\substack{ei_m\in EI,\\ (ei_m,ei_k^j)\in EvConn}} Z_{EI}(ei_m)\vee$$

$$\vee \bigwedge_{i=1}^{N_{CFB}} Z_{\psi_i}(\psi_i)\wedge \bigvee_{\substack{eo_n^x\in EO^x,\\ (eo_n^x,ei_k^i)\in EvConn}} Z_{EO^x}(eo_n^{\tilde{o}})\Rightarrow$$

$$\Rightarrow Z_{EI^j}(ei_k^j)\leftarrow true \mid ei_k^j\in EI^j\},\ j=\overline{1,N_{FB}};$$

б) для случая, когда в составном ФБ одни базисные компонентные ФБ:

$$\{p_{EI^j}^{C,S,2}[k]: Z_\alpha(\alpha)\wedge \bigvee_{\substack{ei_m\in EI,\\ (ei_m,ei_k^j)\in EvConn}} Z_{EI}(ei_m)\vee Z_\mu(\mu)\wedge$$

$$\wedge \bigvee_{\substack{eo_n^x \in EO^x, \\ (eo_n^x, ei_k^i) \in EvConn}} Z_{EO^x}(eo_n^{\tilde{o}}) \Rightarrow Z_{EI^j}(ei_k^j) \leftarrow true \mid ei_k^j \in EI^j\}, \ j = \overline{1, N_{FB}}.$$

Правила изменения выходных событийных переменных строятся на основе правила $p_{EO}^{C,C,1}$ путем добавления в него проверки переменных β_i и ψ_i, поскольку передача сигналов на выход производится только после завершения второй фазы во всех компонентных ФБ:

а) для случая, когда в составном ФБ имеется хотя бы один составной компонентный ФБ:

$$\{p_{EO}^{C,S,1}[k]: (\bigvee_{\substack{ei_m \in EI, \\ (ei_m, eo_k) \in EvConn}} Z_{EI}(ei_m) \vee \bigvee_{\substack{eo_n^x \in EO^x, \\ (eo_n^x, eo_k) \in EvConn}} Z_{EO^x}(eo_n^x)) \wedge$$

$$\wedge \bigwedge_{i=1}^{N_{CFB}} Z_{\psi_i}(\psi_i) \Rightarrow Z_{EO}(eo_k) \leftarrow true \mid eo_k \in EO\};$$

б) для случая, когда в составном ФБ одни базисные компонентные ФБ:

$$\{p_{EO}^{C,S,2}[k]: (\bigvee_{\substack{ei_m \in EI, \\ (ei_m, eo_k) \in EvConn}} Z_{EI}(ei_m) \vee \bigvee_{\substack{eo_n^x \in EO^x, \\ (eo_n^x, eo_k) \in EvConn}} Z_{EO^x}(eo_n^x)) \wedge Z_{\mu}(\mu) \Rightarrow$$

$$\Rightarrow Z_{EO}(eo_k) \leftarrow true \mid eo_k \in EO\}.$$

Поскольку в нашем случае используется синхронный съем данных, то при приеме стартового сигнала α сбрасываются все входные сигналы. Для этих целей может использоваться правило $p_{EI}^{C,C,1}$, определенное для циклической модели.

Сброс выходных событийных переменных компонентных ФБ производится в конце второй фазы:

а) для случая, когда в составном ФБ имеется хотя бы один составной компонентный ФБ:

$$\{p_{EO^j}^{C,S,1}[k]: Z_{EO^j}(eo_k^j) \wedge \bigwedge_{i=1}^{N_{CFB}} Z_{\psi_i}(\psi_i) \Rightarrow$$

$$\Rightarrow Z_{EO^j}(eo_k^j) \leftarrow false \mid eo_k^j \in EO^j\}, j = \overline{1, N_{FB}};$$

б) для случая, когда в составном ФБ одни базисные компонентные ФБ:

$$\{p_{EO^j}^{C,S,2}[k]: Z_{EO^j}(eo_k^j) \wedge Z_{\mu}(\mu) \Rightarrow Z_{EO^j}(eo_k^j) \leftarrow$$

$$\leftarrow false \mid eo_k^j \in EO^j\}, j = \overline{1, N_{FB}}.$$

Выдача выходных данных определяется следующими правилами:

а) для случая, когда в составном ФБ имеется хотя бы один составной компонентный ФБ:

$$\{p_{VOB}^{C,S,1}[k]: \bigwedge_{i=1}^{N_{CFB}} Z_{\psi_i}(\psi_i) \wedge \bigvee_{(eo_k, vob_m)\in OW} (\bigvee_{\substack{ei_j\in EI,\\ (ei_j, eo_k)\in EvConn}} Z_{EI}(ei_j) \vee$$

$$\vee \bigvee_{\substack{eo_n^x\in EO^x,\\ (eo_n^x, eo_k)\in EvConn}} Z_{EO^x}(eo_n^x)) \Rightarrow Z_{VOB}(vob_m) \leftarrow$$

$$\leftarrow Z_{VO}(repr_{VO}(vob_m)) \mid vob_m \in VOB\};$$

б) для случая, когда в составном ФБ одни базисные компонентные ФБ:

$$\{p_{VOB}^{C,S,2}[k]: Z_{\mu}(\mu) \wedge \bigvee_{(eo_k, vob_m)\in OW} (\bigvee_{\substack{ei_j\in EI,\\ (ei_j, eo_k)\in EvConn}} Z_{EI}(ei_j) \vee$$

$$\vee \bigvee_{\substack{eo_n^x\in EO^x,\\ (eo_n^x, eo_k)\in EvConn}} Z_{EO^x}(eo_n^x)) \Rightarrow Z_{VOB}(vob_m) \leftarrow$$

$$\leftarrow Z_{VO}(repr_{VO}(vob_m)) \mid vob_m \in VOB\}.$$

3.12. Модель диспетчера для синхронной модели выполнения

Для демонстрации различных схем построения систем взаимодействующих модулей ФБ и диспетчеров рассмотрим случай реализации диспетчера для двухтактной синхронной модели выполнения ФБ в виде асинхронного модуля. Следует отметить, что основные принципы построения «синхронных» и «асинхронных» диспетчеров остаются одними и теми же, однако в последнем случае необходимо явно отслеживать наступление некоторых событий, например, так называемого «затухания переходных процессов», заключающегося в окончании всех передач сигналов и данных в родительском ФБ.

Множество переменных диспетчера промежуточного уровня для синхронной модели определяется следующим набором:

$$V_D^S = (\alpha, \mu, \beta, \psi, (\alpha_i)_{i=\overline{1,N_{FB}}}, (\mu_i)_{i=\overline{1,N_{CFB}}}, (\beta_i)_{i=\overline{1,N_{FB}}}, (\psi_i)_{i=\overline{N_{CFB}}}, \omega),$$

где α и μ – переменные запуска первой и второй фазы выполнения подсистемы соответственно; β и ψ – переменные окончания выпол-

нения первой и второй фазы выполнения подсистемы соответственно; $(\alpha_i)_{i=\overline{1,N_{FB}}}, (\mu_i)_{i=\overline{1,N_{CFB}}}$ – множества переменных запуска первой и второй фазы выполнения компонентных ФБ соответственно; $(\beta_i)_{i=\overline{1,N_{FB}}}$ и $(\psi_i)_{i=\overline{N_{CFB}}}$ – множество переменных окончания выполнения первой и второй фазы компонентных ФБ соответственно; ω – признак завершения передач в родительском составном ФБ.

Здесь N_{FB} – число компонентных ФБ в подсистеме, N_{CFB} – число компонентных составных ФБ в подсистеме. Под подсистемой в данном случае понимается составной ФБ или субприложение. Для простоты предполагаем, что при нумерации компонентных ФБ внутри подсистемы сначала идут составные компонентные ФБ, а затем базисные компонентные ФБ.

Функции изменения переменных диспетчера промежуточного уровня определяются следующим образом:

$$T_D^S = (t_\alpha, t_\beta, t_\mu, t_\psi, (t_{\alpha_i})_{i=\overline{1,N_{FB}}}, (t_{\beta_i})_{i=\overline{1,N_{FB}}}, (t_{\mu_i})_{i=\overline{1,N_{CFB}}}, (t_{\psi_i})_{i=\overline{1,N_{CFB}}}),$$

где $t_\alpha : [\bigcup_{k=1}^{N_{FB}} Z_{\beta_k}] \to [Z_\alpha]$ – функция сброса переменной α;

$t_\beta : [\bigcup_{k=1}^{N_{FB}} Z_{\beta_k}] \to [Z_\beta]$ – функция установки переменной β;

$t_\mu : [\bigcup_{k=1}^{N_{CFB}} Z_{\psi_k}] \times [Z_\omega] \to [Z_\mu]$ – функция сброса переменной μ;

$t_\psi : [\bigcup_{k=1}^{N_{CFB}} Z_{\psi_k}] \times [Z_\omega] \to [Z_\psi]$ – функция установки переменной ψ.

Если составной ФБ включает только базисные компонентные ФБ, то функция сводится к функции $t_\psi : [Z_\omega] \to [Z_\psi]$;

$t_{\alpha_i} : [Z_\alpha] \times [Z_\omega] \to [Z_{\alpha_i}]$ – функция установки переменной α_i;

$t_{\mu_i} : [Z_\mu] \to [Z_{\mu_i}]$ – функция установки переменной μ_i;

$t_{\beta_i} : [\bigcup_{k=1}^{N_{FB}} Z_{\beta_k}] \to [Z_{\beta_i}]$ – функция сброса переменной β_i;

$t_{\psi_i} : [\bigcup_{k=1}^{N_{CFB}} Z_{\psi_k}] \times [Z_\omega] \to [Z_{\psi_i}]$ – функция сброса переменной ψ_i.

Ниже приводятся правила функционирования диспетчера промежуточного уровня.

Правила для сброса переменной α и установки переменной β соответственно:

$$p_{\alpha}^{1,D,S}: \bigwedge_{k=1}^{N_{FB}} Z_{\beta_k}(\beta_k) \Rightarrow Z_{\alpha}(\alpha) \leftarrow false;$$

$$p_{\beta}^{1,D,S}: \bigwedge_{k=1}^{N_{FB}} Z_{\beta_k}(\beta_k) \Rightarrow Z_{\beta}(\beta) \leftarrow true.$$

В соответствии с данными правилами сброс α и установка β производятся, когда все компонентные ФБ, входящие в состав подсистемы, закончили выполнение первой фазы.

Правила для сброса переменной μ и установки переменной ψ различаются для случаев:

а) подсистема содержит один или несколько составных компонентных ФБ:

$$p_{\mu}^{1,D,S}: \bigwedge_{k=1}^{N_{CFB}} Z_{\psi_k}(\psi_k) \Rightarrow Z_{\mu}(\mu) \leftarrow false;$$

$$p_{\psi}^{1,D,S}: \bigwedge_{k=1}^{N_{CFB}} Z_{\psi_k}(\psi_k) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\psi}(\psi) \leftarrow true;$$

б) подсистема содержит только базисные компонентные ФБ:

$$p_{\mu}^{2,D,S}: Z_{\mu}(\mu) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\mu}(\mu) \leftarrow false;$$

$$p_{\psi}^{2,D,S}: Z_{\mu}(\mu) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\psi}(\psi) \leftarrow true.$$

Правило для установки переменных α_i $(i = \overline{1, N_{FB}})$ представлено ниже:

$$p_{\alpha_i}^{1,D,S}: Z_{\alpha}(\alpha) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\alpha_i}(\alpha_i) \leftarrow true.$$

В соответствии с данными правилами первая фаза выполнения *i*-го компонентного ФБ запускается (в следующем такте) при наличии сигнала запуска α подсистемы в целом. При этом все передачи сигналов в составном ФБ (с входов этого блока) должны быть завершены. Как можно заметить, все компонентные ФБ запускаются одновременно. Следует отметить, что запуск компонентных ФБ будет производиться корректно – при готовых сигналах и данных на их входах, поскольку запуск компонентных ФБ отстает на один такт от действий по съему данных и передачи сигналов на входы компонентных ФБ.

Правило для сброса переменных β_i $(i=\overline{1,N_{FB}})$:

$$p_{\beta_i}^{1,D,S}: \bigwedge_{k=1}^{N_{FB}} Z_{\beta_k}(\beta_k) \Rightarrow Z_{\beta_i}(\beta_i) \leftarrow false.$$

Как можно заметить, переменные β_i сбрасываются одновременно с установкой переменной β.

Правило для установки переменных μ_i $(i=\overline{1,N_{CFB}})$ запуска второй фазы выполнения компонентных ФБ:

$$p_{\mu_i}^{1,D,S}: Z_{\mu}(\mu) \Rightarrow Z_{\mu_i}(\mu_i) \leftarrow true.$$

Данное правило структурно похоже на правило $p_{\alpha_i}^{1,D,S}$, но ожидания завершения передач сигналов (с входов) не требуется.

Правило для сброса переменных ψ_i $(i=\overline{1,N_{CFB}})$ окончания второй фазы выполнения компонентных ФБ:

$$p_{\psi_i}^{1,D,S}: Z_{\omega}(\omega) \wedge \bigwedge_{k=1}^{N_{CFB}} Z_{\psi_k}(\psi_k) \Rightarrow Z_{\psi_i}(\psi_i) \leftarrow false.$$

Данное правило структурно похоже на правило $p_{\beta_i}^{1,D,S}$, но в данном случае требуется ожидание завершения передач сигналов на выходы.

Следует еще раз подчеркнуть, что вторая фаза определена только для составных ФБ. Если в составе подсистемы нет составных компонентных ФБ, то выполнение второй фазы сводится только к передвижению сигналов внутри подсистемы.

Отличие главного диспетчера (обозначим его *D*') от диспетчера промежуточного уровня состоит в том, что он является полностью независимым от других диспетчеров, и соответственно, в нем отсутствуют внешние управляющие сигналы управления фазами выполнения ФБ. Главный диспетчер определяется для сети ФБ самого верхнего уровня. Как правило, это сеть ФБ, размещаемая на ресурсе. Как только главный диспетчер заканчивает первую фазу выполнения, он сразу же приступает к выполнению второй, при этом первая и вторая фазы чередуются. В случае диспетчера верхнего уровня набор переменных V_D^S и набор функций T_D^S сокращаются и принимают следующие виды:

$$V_{D'}^S = ((\alpha_i)_{i=\overline{1,N_{FB}}}, (\mu_i)_{i=\overline{1,N_{CFB}}}, (\beta_i)_{i=\overline{1,N_{FB}}}, (\psi_i)_{i=\overline{N_{CFB}}}, \omega);$$

$$T_{D'}^S = ((t_{\alpha_i})_{i=\overline{1,N_{FB}}}, (t_{\beta_i})_{i=\overline{1,N_{FB}}}, (t_{\mu_i})_{i=\overline{1,N_{CFB}}}, (t_{\psi_i})_{i=\overline{1,N_{CFB}}}).$$

При этом схемы функций будут следующими:

$t_{\alpha_i} : [\bigcup_{i=1}^{N_{FB}} Z_{\beta_i}] \times [\bigcup_{i=1}^{N_{CFB}} Z_{\psi_i}] \times [Z_\omega] \to [Z_{\alpha_i}]$ – функция установки переменной α_i;

$t_{\mu_i} : [\bigcup_{k=1}^{N_{FB}} Z_{\beta_k}] \to [Z_{\mu_i}]$ – функция установки переменной μ_i;

$t_{\beta_i} : [\bigcup_{k=1}^{N_{FB}} Z_{\beta_k}] \times [Z_\omega] \to [Z_{\beta_i}]$ – функция сброса переменной β_i;

$t_{\psi_i} : [\bigcup_{k=1}^{N_{CFB}} Z_{\psi_k}] \times [Z_\omega] \to [Z_{\psi_i}]$ – функция сброса переменной ψ_i.

Ниже приведены правила, реализующие эти функции, для случая, когда в подсистеме имеется хотя бы один составной компонентный ФБ:

$$p_{\alpha_i}^{1,D',S} : \bigwedge_{k=1}^{N_{CFB}} Z_{\psi_k}(\psi_k) \wedge Z_\omega(\omega) \Rightarrow Z_{\alpha_i}(\alpha_i) \leftarrow true;$$

$$p_{\mu_i}^{1,D',S} : \bigwedge_{k=1}^{N_{FB}} Z_{\beta_k}(\beta_k) \Rightarrow Z_{\mu_i}(\mu_i) \leftarrow true;$$

$$p_{\beta_i}^{1,D',S} : \bigwedge_{k=1}^{N_{FB}} Z_{\beta_k}(\beta_k) \wedge Z_\omega(\omega) \Rightarrow Z_{\beta_i}(\beta_i) \leftarrow false;$$

$$p_{\psi_i}^{1,D',S} : \bigwedge_{k=1}^{N_{CFB}} Z_{\psi_k}(\psi_k) \wedge Z_\omega(\omega) \Rightarrow Z_{\psi_i}(\psi_i) \leftarrow false.$$

Если подсистема содержит только базисные компонентные ФБ, то набор правил существенно упрощается – он будет включать правило

$$p_{\alpha_i}^{2,D',S} : \bigwedge_{k=1}^{N_{FB}} Z_{\beta_k}(\beta_k) \wedge Z_\omega(\omega) \Rightarrow Z_{\alpha_i}(\alpha_i) \leftarrow true,$$

а также правила типа $p_{\beta_i}^{1,D',S}$.

Идея и основные моменты в определении операционной семантики синхронных ФБ на основе МАС были представлены в работах [33, 34].

3.13. Семантика последовательной модели выполнения

3.13.1. Особенности модели

Особенности и ограничения предлагаемой формальной модели ФБ заключаются в следующем. Как таковой, отдельный модуль диспетчера отсутствует. При этом функции диспетчирования разнесены по модулям ФБ. Основой диспетчера является глобальная *FIFO*-очередь входных сигналов, предназначенных для обработки (назовем ее для определенности *QUE*). В процессе работы системы ФБ вновь сгенерированные сигналы, предназначенные для посылки на событийные входы ФБ, помещаются в конец данной очереди. Для текущей обработки выбирается сигнал из начала очереди. Какой модуль ответственен за обработку первого элемента очереди *QUE*, распознается на основе уникальных идентификаторов, распределенных каждому из модулей. Поскольку в развернутой системе ФБ существует взаимно однозначное соответствие между модулями и компонентными ФБ, то модулю может быть приписан идентификатор, сопоставленный соответствующему компонентному ФБ. Элемент очереди представляет собой пару типа (idm_j, ei_k), где $idm_j \subseteq ID^M$– идентификатор модуля ФБ, в котором будет производиться обработка входного сигнала ei_k.

Для организации очереди *QUE* используется кольцевой буфер, имеющий два указателя: *iw* – указатель записи и *ir* – указатель чтения. После записи элемента в очередь производится инкремент указателя *iw* по модулю *N*: $iw = (iw + 1)_{\mathrm{mod}\ N}$, где *N* – размер буфера. После выборки элемента из очереди производится инкремент указателя *ir*: $ir = (ir + 1)_{\mathrm{mod}\ N}$. Величина *N* выбирается в зависимости от сложности моделируемой системы. Очередь пуста, если $iw = ir$. Для представления очереди *QUE* используется два набора переменных: *QFB* – набор переменных для хранения идентификаторов модулей ФБ и *QEI* – набор переменных для хранения идентификаторов входных событий.

В модели используется диспетчирование на уровне входных событий ФБ (альтернативой является диспетчирование на уровне выходных событий). Планирование нового события может быть представлено как включение информации о событийном входе компонентного ФБ или оболочки составного ФБ, на который поступил сигнал, в конец очереди *QUE*.

При выполнении базисных ФБ используется принцип одиночного прогона ФБ (*Single Run*). Базисный ФБ отрабатывает полно-

стью, прежде чем начнется проводка сгенерированных им выходных сигналов. Сгенерированные сигналы помещаются в специальный список *EOL* в хронологическом порядке. Поскольку модель выполнения последовательная, список *EOL* единственен для всей системной модели – с ним работает тот ФБ, который выполняется. Таким образом, МБФБ взаимодействует с внешней средой через очередь *QUE* – по входам и через список *EOL* – по выходам.

Составной ФБ представляется в виде контейнера, а не в виде единой сущности. Диспетчируется выполнение входного интерфейса составного ФБ, в то время как выполнение выходного интерфейса не диспетчируется. МСФБ взаимодействует с внешней средой через очередь *QUE* – по входам и через выходные событийные переменные – по выходам.

При проводке сигнала от источника к потребителю через цепочку выходных интерфейсов используется принцип «горячей картошки» [131], т.е. сигнал передается через выходной интерфейс немедленно, как только он появляется. Это говорит о том, что работа выходных интерфейсов не диспетчируется. Проводка сигнала через последовательность выходных интерфейсов прерывается, как только сигнал достигает входного интерфейса.

Считается, что каждый источник сигнала имеет только одного потребителя, т.е. неявное («аппаратное») расщепление сигнала не используется. Для расщепления сигнала можно использовать ФБ *E_SPLIT*. Также предполагается, что у компонентных составных ФБ нет висячих выходов. Эти выходы могут быть предварительно удалены.

В модельной реализации используются глобальные данные и переменные сложной структуры. Работа с глобальными данными может быть реализована на основе механизма передачи параметров.

3.13.2. Модель базисного ФБ

Определение схемы модели

Модуль базисного ФБ (МБФБ) для последовательной модели выполнения определяется двойкой:

$$M_B^G = (Synt_B, Sem_B^G),$$

где $Synt_B$ – синтаксическая часть описания, определяющая структуру и алгоритмы ФБ (общая для всех моделей); Sem_B^G – семантическая часть описания.

Семантическая часть описания МБФБ для последовательной модели выполнения есть кортеж

$$Sem_B^G = (VRT_B^G, T_B^G),$$

где VRT_B^G – набор переменных и констант времени выполнения; T_B^G – набор функций переходов МБФБ.

Набор переменных и констант времени выполнения определяется кортежем

$$VRT_B^G = (idm, QFB, QEI, ir, \sigma, EOL, ier, iew, VIB, VOB, Q, DomQ, NA, NI, \beta),$$

где *idm* – уникальный идентификатор МБФБ; $QFB = (QFB_1, QFB_2, \ldots, QFB_N)$ – набор переменных для организации очереди и хранения идентификаторов модулей ФБ; $Dom(QFB) = ID^M$, где ID^M – множество идентификаторов модулей ФБ; $QEI = (QEI_1, QEI_2, \ldots, QEI_N)$ – набор переменных для организации очереди и для хранения (идентификаторов) входных событий, $Dom(QEI) = EI$; *ir* – переменная, в которой хранится номер первой ячейки очереди *QUE* (указатель чтения очереди); σ – (глобальная) переменная разрешенности работы модулей ФБ, $Dom(\sigma) = \{true, false\}$. Модуль ФБ может начать выполнение, если данный признак установлен. В конце выполнения модуля ФБ признак сбрасывается; $EOL = \{EOL_1, EOL_2, ..., EOL_M\}$ – (глобальный) набор переменных для организации списка выходных событий модуля ФБ, где *M* – максимальный размер списка, определяемый исходя из максимального числа сигналов, выданных за одно выполнение ФБ. Как правило, $M \leq N_{EO}$, $Dom(EOL) = EO$; *ier* и *iew* – указатели чтения и записи в список *EOL* соответственно, $Dom(ier) = Dom(iew) = \{0, 1, \ldots, M\}$.

Компоненты кортежа *VIB*, *VOB*, *Q*, *NA*, *NI* и β имеют такой же смысл, как и в кортеже Sem_B для базовой модели МБФБ.

Следует отметить, что в кортеже VRT_B^G отсутствует *S* – переменная текущего состояния *OSM*-машины, собственно, как и сама *OSM*-машина в явном виде. Это связано, прежде всего, с тем, что ФБ в последовательной модели выполняются последовательно, любые конфликты в выполнении блоков исключены. Тем не менее некоторые состояния *OSM*-машины неявно представляются с помощью комбинации значений некоторых переменных *NA* и *NI* времени выполнения.

Введем следующие обозначения: $ENM = [Z_\sigma] \times [Z_{idm}] \times [Z_{QFB}] \times \times [Z_{ir}]$ – комбинация функций, определяющих условия для запуска МБФБ; $PTR = [Z_Q] \times [Z_{NA}] \times [Z_{NI}]$ – комбинация функций, опреде-

ляющих положение «указателей» выполнения ФБ – идентификатора *EC*-состояния, номера выполняемой *EC*-акции и номера выполняемого шага алгоритма.

Набор функций переходов МБФБ определяется как

$$T_B^G = (t_{QEI}, t_{VI}, t_{VO}, t_{VV}, t_{VOB}, t_{EOL}, t_{eiw}, t_Q, t_{NA}, t_{NI}, t_\beta),$$

$t_{QEI} : [Z_{QEI}] \times PTR \times [Z_{ir}] \times [Z_V] \to [Z_{QEI}]$ – функция сброса активного использованного события в очереди *QUE*;

$t_{VI} : [Z_{VIB}] \times ENM \to [Z_{VI}]$ – функция изменения входных переменных (в результате съема данных);

$t_{VO} : [Z_V] \times PTR \to [Z_{VO}]$ – функция изменения выходных переменных (в результате выполнения алгоритма);

$t_{VV} : [Z_V] \times PTR \to [Z_{VV}]$ – функция изменения внутренних переменных (в результате выполнения алгоритма).

Следует отметить, что при определении функций t_{VO} и t_{VV} кортеж *ENM* не используется.

$t_{VOB} : [Z_{VO}] \times PTR \to [Z_{VOB}]$ – функция изменения выходных буферов (в результате выдачи данных);

$t_{EOL} : [Z_{EOL}] \times [Z_{iew}] \times PTR \to [Z_{EOL}]$ – функция записи в конец списка *EOL* идентификатора выходного сигнала;

$t_{iew} : [Z_{iew}] \times PTR \to [Z_{iew}]$ – функция инкремента указателя записи списка *EOL*;

$t_Q : PTR \times [Z_{QEI}] \times [Z_{ir}] \times [Z_V] \to [Z_Q]$ – функция переходов *EC*-состояний;

$t_{NA} : PTR \times [Z_{QEI}] \times [Z_{ir}] \times [Z_V] \to [Z_{NA}]$ – функция изменения указателя *EC*-акций;

$t_{NI} : PTR \times [Z_{QEI}] \times [Z_{ir}] \times [Z_V] \to [Z_{NI}]$ – функция изменения указателя инструкций в алгоритме;

$t_\beta : ENM \times PTR \times [Z_V] \to [Z_\beta]$ – функция сброса признака разрешенности выполнения МБФБ.

Определение динамики модели

Работа модуля базисного ФБ неформально может быть представлена следующим образом. Сначала ожидается появление в первом элементе очереди *QUE* идентификатора данного модуля (при действующем сигнале разрешения σ). После этого начинается обработка входного сигнала, который указан в первом элементе

очереди *QUE*. Это соответствует выполнению ФБ. В результате работы ФБ формируется список выходных событий *EOL*. После завершения выполнения ФБ устанавливается признак окончания работы ФБ (сбрасывается переменная σ).

Ниже функционирование МБФБ представлено формально в виде правил изменения функций значений переменных.

Определим условие перехода в диаграмме *ECC* для последовательной модели выполнения: а) для триггерных *EC*-переходов $(q_i, ei_k, q_j) \in ECTran$:

$$TranCond'(q_i, ei_k, q_j, Z_{QEI}, Z_{ir,} Z_V) \triangleq$$

$$\triangleq Z_{QEI}(QEI_{ir}) = ei_k \wedge GuardCond(q_i, ei_k, q_j, Z_V),$$

где $QEI_{ir} \triangleq QEI_{Z_{ir}(ir)}$ − переменная из набора *QEI*, индекс которой определяется значением указателя *ir*;

б) для нетриггерных *EC*-переходов $(q_i, \varepsilon, q_j) \in ECTran$:

$$TranCond'(q_i, \varepsilon, q_j, Z_{QEI}, Z_{ir,} Z_V) \triangleq GuardCond(q_i, \varepsilon, q_j, Z_V).$$

Правило сброса активного события в очереди *QUE*:

$$p_{QEI}^{B,G,1} : Z_{NA}(NA) = 0 \wedge Z_{NI}(NI) = 0 \wedge$$

$$\wedge \bigvee_{(q_i, x, q_j) \in ECTran} (Z_Q(Q) = q_i \wedge TranCond'(q_i, x, q_{j,} Z_{QEI}, Z_{ir}, Z_V)) \Rightarrow$$

$$\Rightarrow Z_{QEI}(QEI_{ier}) \leftarrow \varepsilon,$$

где ε − признак отсутствия сигнала. Сброс активного события производится для того, чтобы оно не могло использоваться при оценке других *EC*-переходов при выполнении ФБ как *Single Run*.

Правила изменения входных переменных ФБ в результате съема данных:

$$\{p_{VI}^{B,G,1}[j] : Z_\sigma(\sigma) \wedge Z_{QFB}(QFB_{ir}) = idm \wedge \bigvee_{(ei_k, vi_m) \in IW} (Z_{QEI}(QEI_{ir}) = ei_k) \Rightarrow$$

$$\Rightarrow Z_{VI}(vi_j) \leftarrow Z_{VIB}(vib_j) \mid vi_j \in VI\}.$$

Изменения выходных переменных ФБ в результате выполнения алгоритмов определяются с помощью правил $p_{VO}^{B,1}$ для базовой модели базисного ФБ. Аналогичные правила могут быть составлены для переменных из множества *VV*.

Правила выдачи выходных сигналов в список *EOL*:

$$\{p_{EOL}^{B,G,1}[k] : putoutEO'_k \Rightarrow Z_{EOL}(EOL_{iew}) \leftarrow eo_k \mid eo_k \in EO\},$$

где $putoutEO'_k \triangleq Z_{NI}(NI) = 0 \wedge \bigvee_{q_i \in Q^{eo_k}, j \in NA^{q_i, eo_k}} (Z_Q(Q) =$

$= q_i \wedge Z_{NA}(NA) = j)$ – условие выдачи выходных сигналов.

Изменения выходных буферов данных осуществляются в соответствии с правилами $p_{VOB}^{B,1}$ для базовой модели базисного ФБ.

Правила изменения значения указателя *iew* записи в буфер *EOL*:

$$p_{iew}^{B,G,1} : \bigvee_{eo_k \in EO} putoutEO'_k \Rightarrow Z_{iew}(iew) \leftarrow Z_{iew}(iew) + 1.$$

Правила изменения *EC*-состояний сгруппированы в приоритетную группу:

$$< p_Q^{B,G,1}[i,x,j] : Z_Q(Q) = q_i \wedge Z_{NA}(NA) = 0 \wedge Z_{NI}(NI) = 0 \wedge$$
$$\wedge TranCond'(q_i, x, q_{j,} Z_{QEI}, Z_{ir}, Z_V) \Rightarrow Z_Q(Q) \leftarrow q_j \mid (q_i, x, q_j) \in$$
$$\in ECTranOrd > .$$

Следует обратить внимание, что переход из одного *EC*-состояния в другое возможен только тогда, когда счетчики *NA* и *NI* равны нулю. Это соответствует состоянию *s*1 *OSM*-машины [163].

Правила изменения счетчика *EC*-акций *NA*:

$$p_{NA}^{B,G,1} : Z_{NA}(NA) = 0 \wedge Z_{NI}(NI) = 0 \wedge$$
$$\wedge \bigvee_{(q_i, x, q_j) \in ECTran} (Z_Q(Q) = q_i \wedge TranCond'(q_i, x, q_{j,} Z_{QEI}, Z_{ir}, Z_V)) \Rightarrow$$
$$\Rightarrow Z_{NA}(NA) \leftarrow 1;$$
$$p_{NA}^{B,G,2} : Z_{NI}(NI) = 0 \wedge \bigvee_{q_i \in DomQ} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) = N_{NA}^{q_i}) \Rightarrow$$
$$\Rightarrow Z_{NA}(NA) \leftarrow 0;$$
$$p_{NA}^{B,G,3} : Z_{NI}(NI) = 0 \wedge \bigvee_{q_i \in DomQ} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) < N_{NA}^{q_i}) \Rightarrow$$
$$\Rightarrow Z_{NA}(NA) \leftarrow Z_{NA}(NA) + 1.$$

Изменение счетчика шагов алгоритмов *NI* производится в соответствии с правилами $p_{NI}^{B,G,1}, p_{NI}^{B,2}$ è $p_{NI}^{B,3}$. Последние два правила были определены для базовой модели:

$$p_{NI}^{B,G,1} : Z_{NA}(NA) = 0 \wedge Z_{NI}(NI) = 0 \wedge \bigvee_{(q_i, x, q_j) \in ECTran} (Z_Q(Q) =$$
$$= q_i \wedge TranCond'(q_i, x, q_{j,} Z_{QEI}, Z_{ir}, Z_V)) \Rightarrow Z_{NI}(NI) \leftarrow 1.$$

Правило для установки переменной окончания выполнения ФБ β:

$$p_{\beta}^{B,G,1} : Z_{NA}(NA) = 0 \wedge Z_{NI}(NI) = 0 \wedge$$

$$\wedge \bigvee_{q_i \in DomQ} (Z_Q(Q) = q_i \wedge \bigwedge_{(q_i, x, q_j) \in ECTran} \overline{TranCond'(q_i, x, q_j, Z_{QEI}, Z_{ir}, Z_V)} \Rightarrow$$

$$\Rightarrow Z_{\beta}(\beta) \leftarrow true.$$

Согласно этому правилу, МБФБ заканчивает работу, когда в текущем *EC*-состоянии закончено выполнение всех *EC*-акций, и из текущего *EC*-состояния нет разрешенных *EC*-переходов.

3.13.3. Модель составного ФБ

Определение схемы модели

МСФБ для последовательной модели выполнения определяется как

$$M_C^G = (Synt_C, Sem_C^G),$$

где $Synt_C$ – синтаксическая часть описания (общая для всех МСФБ); $Sem_C^G = (VRT_C^G, T_C^G)$ – семантическая часть описания для последовательной модели выполнения.

Набор переменных и констант времени выполнения определяется кортежем

$$VRT_C^G = (idm, QFB, QEI, iw, ir, EOL, iew, ier, VIB, VOB, \sigma, \beta, \psi),$$

где *idm* – уникальный идентификатор МСФБ; *iw* – переменная, в которой хранится номер первой свободной ячейки, следующей за последней занятой ячейкой очереди *QUE* (указатель записи очереди); ψ – (глобальная) переменная управления автоматической передачей сигналов через цепочки выходных интерфейсов.

Переменные *QFB*, *QEI*, *ir*, *EOL*, *ier*, *iew*, *VIB*, *VOB*, σ и β имеют такой же смысл, что и в наборе VRT_B^G.

Определим $SCHED = SCHED1 \times SCHED2 \times SCHED3$ – комбинацию функций, определяющих планирование нового события в очереди *QUE*, где

$SCHED1 = [Z_{QFB}] \times [Z_{QEI}] \times [Z_{ir}] \times [Z_{\sigma}]$ – комбинация функций, определяющих планирование нового события в очереди *QUE* вследствие обработки входного события составного ФБ;

$SCHED2 = [Z_{\beta}] \times [Z_{\psi}] \times [Z_{EOL}] \times [Z_{ier}] \times [Z_{iew}] \times [Z_{QFB}] \times [Z_{ir}]$ – комби-

нация функций, определяющих планирование нового события в очереди *QUE* вследствие обработки выходного события компонентного базисного ФБ (из списка *EOL*);

$SCHED3 = [\bigcup_{i=1}^{N_{CFB}} Z_{EO^i}]$ – комбинация функций, определяющих планирование нового события в очереди *QUE* вследствие обработки выходного события компонентного составного ФБ.

Тогда набор функций переходов МБФБ можно определить как

$$T_C{}^G = (t_{QFB}, t_{QEI}, t_{iw}, t_{ir}, t_{iew}, t_{ier}, t_{VI}, t^1{}_{EO}, t^2{}_{EO}, t^3{}_{EO}, t^1{}_{VOB}, t^2{}_{VOB}, t^3{}_{VOB}, t_\sigma, t_\beta, t_\psi),$$

где $t_{QFB} : SCHED \to [Z_{QFB}]$ – функция изменения очереди *QFB*;

$t_{QEI} : SCHED \to [Z_{QEI}]$ – функция изменения очереди *QEI*;

$t_{iw} : SCHED \times [Z_{iw}] \to [Z_{iw}]$ – функция изменения указателя записи *iw* в очередь *QUE*;

$t_{ir} : [Z_{QFB}] \times [Z_{ir}] \times [Z_\sigma] \times [Z_\beta] \to [Z_{ir}]$ – функция изменения указателя чтения *ir* из очереди *QUE*;

$t_{ier} : [Z_{ier}] \times [Z_{iew}] \times [Z_\psi] \times [Z_\beta] \to [Z_{ier}]$ – функция изменения указателя чтения списка выходных событий *EOL*;

$t_{iew} : [Z_\beta] \times [Z_\psi] \times [Z_{ier}] \times [Z_{iew}] \to [Z_{iew}]$ – функция изменения указателя записи списка выходных событий *EOL*;

$t_{VI} : [Z_{VIB}] \times [Z_\sigma] \times [Z_{QFB}] \times [Z_{QEI}] \times [Z_{ir}] \to [Z_{VI}]$ – функция изменения входных переменных (в результате съема данных);

$t^1_{EO} : SCHED1 \to [Z_{EO}]$ – функция установки выходных событийных переменных модуля в результате передачи сигналов на его выходы из входных событийных переменных;

$t^2_{EO} : SCHED2 \to [Z_{EO}]$ – функция установки выходных событийных переменных модуля в результате передачи сигналов на его выходы из списка *EOL* компонентного базисного ФБ;

$t^3_{EO} : SCHED3 \to [Z_{EO}]$ – функция установки выходных событийных переменных модуля в результате передачи сигналов на его выходы с выходных переменных компонентного составного ФБ;

$t^1_{VOB} : [\bigcup_{i=1}^{N_{FB}} Z_{VO_i}] \times SCHED1 \to [Z_{VOB}]$ – функция изменения выходных буферов в результате «сквозной» передачи сигнала с входа на выход составного ФБ;

$t^2_{VOB} : [\bigcup_{i=1}^{N_{FB}} Z_{VO_i}] \times SCHED2 \to [Z_{VOB}]$ – функция изменения выходных буферов в результате выдачи данных из компонентного базисного ФБ;

$t^3_{VOB} : [\bigcup_{i=1}^{N_{FB}} Z_{VO_i}] \times SCHED3 \to [Z_{VOB}]$ – функция изменения выходных буферов в результате выдачи данных из компонентного составного ФБ;

$t_\psi : SCHED \to [Z_\psi]$ – функция изменения признака распространения сигнала через цепочки выходных интерфейсов;

$t_\sigma : [Z_{ier}] \times [Z_{iew}] \times [Z_\beta] \to [Z_\sigma]$ – функция установки признака разрешенности работы ФБ;

$t_\beta : [Z_{ier}] \times [Z_{iew}] \times [Z_\beta] \times [Z_\psi] \to [Z_\beta]$ – функция сброса признака окончания выполнения ФБ.

Определение динамики модели

Для краткости определения правил введем следующие условия для антецедентов правил:

1) условие планирования события непосредственно по входному сигналу МСФБ:

$$SchedOnEI_k \triangleq Z_{EI}(ei_k);$$

2) условие планирования события по «плановому» входному событию МСФБ:

$$SchedOnQEI \triangleq Z_\sigma(\sigma) \wedge (Z_{QFB}(QFB_{ir}) =$$

$$= idm) \wedge postEv(Z_{QEI}(QEI_{ir})) \in \bigcup_{i=1}^{N_{FB}} EI^{i,(idm)},$$

где $EI^{i,(idm)}$ – множество входных событийных переменных *i*-го компонентного ФБ, входящего в состав МСФБ с идентификатором *idm*. Здесь и далее для уточнения, если в этом возникает необходимость, на месте верхнего индекса в круглых скобках будет указываться идентификатор модуля, к которому относится объект;

3) условие планирования события по выходному сигналу МБФБ:

$$SchedOnEOL \triangleq (Z_{QFB}(QFB_{ir}) = idm) \wedge Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) <$$

$$< Z_{iew}(iew)) \wedge postEv(Z_{EOL}(EOL_{ier})) \in \bigcup_{i=1}^{N_{FB}} EI^{i,(idm)};$$

4) условие появления выходного сигнала составного ФБ по плановому входному событию МСФБ:

$$AriseEOonQEI \triangleq Z_{\sigma}(\sigma) \wedge (Z_{QFB}(QFB_{ir}) =$$

$$= idm) \wedge postEv(Z_{QEI}(QEI_{ir})) \in EO^{(idm)};$$

5) условие появления выходного сигнала составного ФБ по выходному событию МБФБ:

$$AriseEOonEOL \triangleq (Z_{QFB}(QFB_{ir}) = idm) \wedge Z_{\beta}(\beta) \wedge \overline{Z_{\psi}(\psi)} \wedge$$

$$\wedge (Z_{ier}(ier) < Z_{iew}(iew)) \wedge postEv(Z_{EOL}(EOL_{ier})) \in EO^{(idm)}.$$

В приведенных выше условиях использовались следующие обозначения:

$postEv : EI \cup \bigcup_{i=1}^{N_{FB}} EO^i \to EO \cup \bigcup_{i=1}^{N_{FB}} EI^i$ – функция, ставящая в соответствие каждому событию его преемника в составном ФБ. Этот преемник единственен ввиду принятых выше предположений. Данная функция, по сути, – не что иное, как представление отношения *EvConn*, введенное для удобства. Следует отметить, что данная функция определяется априори по описанию ФБ.

Ниже функционирование МСФБ представлено формально в виде правил изменения функций значений переменных.

Правила $p_{QEI}^{C,G,1}, p_{QEI}^{C,G,2}, p_{QEI}^{C,G,3}, p_{QFB}^{C,G,1}, p_{QFB}^{C,G,2}, p_{QFB}^{C,G,3}$ относятся к планированию нового события. Правила для планирования входного события компонентного ФБ:

$$p_{QEI}^{C,G,1} : SchedOnQEI \Rightarrow Z_{QEI}(QEI_{iw}) \leftarrow postEv(Z_{QEI}(QEI_{ir}));$$

$$p_{QEI}^{C,G,2} : SchedOnEOL \Rightarrow Z_{QEI}(QEI_{iw}) \leftarrow postEv(Z_{EOL}(EOL_{ier}));$$

$$p_{QFB}^{C,G,1} : SchedOnQEI \Rightarrow Z_{QFB}(QFB_{iw}) \leftarrow postM(Z_{QEI}(QEI_{ir}));$$

$$p_{QFB}^{C,G,2} : SchedOnEOL \Rightarrow Z_{QFB}(QFB_{iw}) \leftarrow postM(Z_{EOL}(EOL_{ier})),$$

где $postM : EI \cup \bigcup_{i=1}^{N_{FB}} EO^i \to IDM \cup \{fb_0\}$ – функция, ставящая в соответствие каждому выходному событию идентификатор целевого ФБ. Здесь значение fb_0 соответствует тем источникам сигналов, которые связаны с событийными выходами составного ФБ.

Первые два вышеприведенных правила добавляют в конец очереди *QUE* событийные входы компонентных ФБ, а последние два – идентификаторы ФБ-обработчиков этих входных сигналов:

$$\{p_{QEI}^{C,G,3}[k]: SchedOnEI_k \Rightarrow Z_{QEI}(QEI_{iw}) \leftarrow ei_k \mid ei_k \in EI\};$$

$$\{p_{QFB}^{C,G,3}[k]: SchedOnEI_k \Rightarrow Z_{QFB}(QFB_{iw}) \leftarrow idm \mid ei_k \in EI\}.$$

Данные группы правил добавляют в конец очереди *QUE*, соответственно, событийные входы составного ФБ и собственный идентификатор модуля ФБ (*idm*), который будет обрабатывать этот сигнал. Следует заметить, что путем подобного планирования прерывается прохождение сигнала через входной интерфейс составного ФБ. При выборе варианта модели выполнения, в котором не требуется диспетчирование выполнения входных интерфейсов составных ФБ, подобное планирование должно быть исключено.

Правило инкремента указателя *iw* записи в очередь *QUE*:

$$p_{iw}^{C,G,1}: (SchedOnEI \vee SchedOnQEI \vee SchedOnEOL) \wedge (Z_{iw}(iw) < \\ < N_{QUE}) \Rightarrow Z_{iw}(iw) \leftarrow Z_{iw}(iw) + 1;$$

$$p_{iw}^{C,G,2}: (SchedOnEI \vee SchedOnQEI \vee SchedOnEOL) \wedge (Z_{iw}(iw) = \\ = N_{QUE}) \Rightarrow Z_{iw}(iw) \leftarrow 1,$$

где N_{QUE} обозначена мощность множеств *QFB* и *QEI*.

Правило инкремента указателя *ir* чтения из очереди *QUE*:

$$p_{ir}^{C,G,1}: Z_{\beta}(\beta) \wedge \overline{Z_{\psi}(\psi)} \wedge (Z_{ier}(ier) = Z_{iew}(iew)) \wedge (Z_{ir}(iw) < N_{QUE}) \Rightarrow \\ \Rightarrow Z_{ir}(ir) \leftarrow Z_{ir}(ir) + 1;$$

$$p_{ir}^{C,G,2}: Z_{\beta}(\beta) \wedge \overline{Z_{\psi}(\psi)} \wedge (Z_{ier}(ier) = Z_{iew}(iew)) \wedge (Z_{ir}(ir) = N_{QUE}) \Rightarrow \\ \Rightarrow Z_{ir}(ir) \leftarrow 1.$$

Правила для посылки сигналов на событийные выходы модуля:

$$p_{EO}^{C,G,1}: AriseEOonQEI \Rightarrow Z_{EO}(postEv(Z_{QEI}(QEI_{ir}))) \leftarrow true;$$

$$p_{EO}^{C,G,2}: AriseEOonEOL \Rightarrow Z_{EO}(postEv(Z_{EOL}(EOL_{ier}))) \leftarrow true;$$

$$\{p_{EO}^{C,G,3}[j]: Z_{EO}(eo_i^k) \Rightarrow Z_{EO}(eo_j) \leftarrow true \mid eo_j \in EO, (eo_i^k, eo_j) \in EvConn\}.$$

Приведенные правила реализуют функции t^1_{EO}, t^2_{EO}, t^3_{EO} соответственно. Последнее из них представляет передачу сигнала через выходной интерфейс составного компонентного ФБ.

Правила для изменения входных переменных *VI* в результате съема данных:

$$\{p_{VI}^{C,G,1}[m]: (SchedOnQEI \vee AriseEOonQEI) \wedge \bigvee_{(ei_k, vi_m) \in IW} Z_{QEI}(QEI_{ir}) = ei_k \Rightarrow \\ \Rightarrow Z_{VI}(vi_m) \leftarrow Z_{VIB}(vib_m) \mid vi_m \in VI\}.$$

Правила изменения выходных буферов *VOB* в результате выдачи данных:

$$\{p_{VOB}^{C,G,1}[m]: \bigvee_{(eo_k, vob_m)\in OW} (Z_{QFB}(QFB_{ir}) = idm \wedge (\bigvee_{(eo_n^x, eo_k)\in EvConn} Z_{EO^x}(eo_n^x) \vee$$

$$\vee Z_\sigma(\sigma) \wedge \bigvee_{(ei_j, eo_k)\in EvConn} Z_{QEI}(QEI_{ir}) = ei_j \vee$$

$$\vee Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) < Z_{iew}(iew)) \wedge \bigvee_{(eo_i^{x,(idm)}, eo_k)\in EvConn} Z_{EOL}(EOL_{ier}) =$$

$$= eo_i^{x,(idm)})) \Rightarrow Z_{VOB}(vob_m) \leftarrow Z_{VO}(repr_{VO}(vob_m)) \mid vob_m \in VOB\}.$$

Правила изменения указателя чтения *ier* из списка *EOL*:

$$p_{ier}^{C,G,1}: Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) < Z_{iew}(iew)) \Rightarrow Z_{ier}(ier) \leftarrow Z_{ier}(ier) + 1;$$

$$p_{ier}^{C,G,2}: Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) = Z_{iew}(iew)) \Rightarrow Z_{ier}(ier) \leftarrow 1.$$

Правило сброса указателя записи *iew* в списке *EOL*:

$$p_{iew}^{C,G,1}: Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) = Z_{iew}(iew)) \Rightarrow Z_{iew}(iew) \leftarrow 1.$$

Правило установки признака разрешенности σ:

$$p_\sigma^{C,G,1}: Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) = Z_{iew}(iew)) \Rightarrow Z_\sigma(\sigma) \leftarrow true.$$

Правило сброса признака окончания работы ФБ β:

$$p_\beta^{C,G,1}: Z_\beta(\beta) \wedge \overline{Z_\psi(\psi)} \wedge (Z_{ier}(ier) = Z_{iew}(iew)) \Rightarrow Z_\beta(\beta) \leftarrow false.$$

Следует отметить, что установка признака σ и сброс признака β выполняются как одно *неделимое* действие.

Правило установки признака автоматического распространения сигнала через цепочки выходных интерфейсов ψ:

$$p_\psi^{C,G,1}: AriseEOonEOL \Rightarrow Z_\psi(\psi) \leftarrow true;$$

$$p_\psi^{C,G,2}: SchedOnEI \Rightarrow Z_\psi(\psi) \leftarrow false.$$

Согласно этим правилам, посланный через выходные интерфейсы сигнал будет считаться принятым, когда он достигнет какого-либо входного интерфейса.

3.14. Формальная модель системы ФБ в виде системы переходов состояний

Ниже представлен другой подход к формализации семантики ФБ – на основе модели Крипке [55], когда определяются глобаль-

ные состояния и глобальные переходы (далее просто переходы), в которых явно определены условия перехода и значения всех переменных после срабатывания перехода. Число подобных переходов, возможно, будет отличаться от числа элементарных правил первой модели как в большую, так и в меньшую сторону в зависимости от характера решаемой задачи.

Как было отмечено выше, развернутую систему ФБ можно представить кортежем *FTree* (см. п. 2.2). Модель развернутой системы ФБ *MS* будем определять следующим образом:

MS= (*SM*$_B$, *SM*$_C$, *VarMap*),

где $SM_B = \{M_B^1, M_B^1, ..., M_B^{N_{MBFB}}\}$ – множество МБФБ;

$SM_C = \{M_C^1, M_C^1, ..., M_C^{N_{MCFB}}\}$ – множество МСФБ;

VarMap – множество отображений переменных модулей, представляющих, по сути, связь формальных и фактических параметров родительских/дочерних модулей.

Поскольку в модель системы ФБ входят модули разных сортов, то определим их состояния отдельно. Состояние МБФБ определяется как

$$S_B = (Z_{EI}, Z_{EO,} Z_{VI}, Z_{VO}, Z_{VOB}, Z_{VV}, Z_Q, Z_S, Z_{NA}, Z_{NI}, Z_\alpha, Z_\beta),$$

где смысл используемых в кортеже функций значений переменных был объяснен выше.

Состояние МСФБ определяется шестеркой:

$$S_C = (Z_{EI}, Z_{EO,} Z_{VI}, Z_{VOB}, Z_\alpha, Z_\beta).$$

Следует отметить, что в состояние S_C не включены значения выходных переменных ФБ, поскольку по принятому выше соглашению эти переменные заменены на их представителей, которыми являются выходные буфера компонентных ФБ.

Состояние модели развернутой системы ФБ определяется следующим образом:

$$S = (\prod_{i=1}^{N_{MBFB}} S_B^i, \prod_{j=1}^{N_{MCFB}} S_C^j),$$

где S_B^i и S_C^j – состояния *i*-го МБФБ и *j*-го МСФБ, входящих в составной ФБ, соответственно.

В данном разделе ограничимся рассмотрением систем ФБ, функционирующих согласно *циклической* модели выполнения. Это-

го будет достаточно, чтобы понять принципы построения формальных моделей подобного класса. Поскольку в рамках циклической модели выполнения можно выделить несколько частных подмоделей, примем следующие ограничения, определяющие в итоге одну из них: 1) используется синхронный съем данных; 2) используется правило сброса входных сигналов *DelEI1*, согласно которому после выбора активного сигнала сбрасываются все входные сигналы; 3) проводка сигналов с выходов компонентных ФБ осуществляется в соответствии с принципом «горячей картошки».

Переходы модели системы ФБ для циклической дисциплины выполнения определяются как

$$R = T^B_{FB} \cup T^B_{SD} \cup T^B_{EX} \cup T^B_{CA} \cup T^B_{CS} \cup T^B_{CF} \cup T^B_{ES} \cup T^C_{TC} \cup T^C_{TO} \cup T^D_{SF} \cup T^D_{PF} \cup T_{\varnothing},$$

где T^B_{FB} – множество переходов типа «Срабатывание *EC*-перехода»; T^B_{SD} – множество переходов типа «Синхронный съем данных в базисном ФБ»; T^B_{EX} – множество переходов типа «Выполнение шага алгоритма»; T^B_{CA} – множество переходов типа «Завершение выполнения *EC*-акций»; T^B_{CS} – переход «Завершение выполнения *EC*-состояния»; T^B_{CF} – переход «Завершение выполнения базисного ФБ»; T^B_{ES} – переход «Пустой запуск базисного ФБ»; T^C_{TC} – множество переходов типа «Синхронный съем данных в составном ФБ»; T^C_{TO} – переход «Передача и выдача сигналов составным ФБ»; T^D_{SF} – переход «Запуск компонентного ФБ»; T^D_{PF} – переход «Завершение выполнения составного ФБ»; $T_{\varnothing}$ – пустой переход, введенный для случая, когда некоторое состояние $s \in S$ не имеет последователя. В этом случае имеет место $R(s,s)$. Это гарантирует то, что отношение переходов в модели Крипке всегда будет тотальным.

Переходы с верхним индексом «*B*» соответствуют базисным ФБ, с индексом «*C*» – составным ФБ, а с индексом «*D*» – диспетчеру. В дальнейшем для унификации изложения всего материала переходы будем представлять в виде продукционных правил по образу и подобию тех, которые были использованы для представления первой формальной модели. Но в отличие от правил первой модели правила второй модели будут включать множество параллельно выполняемых действий. При разделении действий в правиле будет использоваться знак «точка с запятой». Для представления операции параллельного выполнения для группы однотипных действий ис-

пользуется знак Ξ. Общий вид правила: $p_{id}^{m} : c \Rightarrow a_1; a_2; ...a_n$, где c – условие применения правила, $a_1,a_2,\ldots,a_n$ – действия по изменению переменных, id – идентификатор типа перехода, m – модификатор идентификатора типа перехода, включающий обозначение сорта ФБ и номер правила в типе.

Использование концепции глобальных состояний и переходов структуры Крипке вносит определенные сложности в описание иерархической по своей природе системы, даже если она развернута и приведена к одноуровневому представлению. В одноуровневой модели тоже необходимо учитывать логическое разбиение системы на подсистемы в соответствии с первоначальной иерархией. Это диктуется, во-первых, тем, что модели выполнения ФБ ориентированы на работу со структурированной иерархической модульной системой ФБ, а не с «плоским» ее представлением. В дальнейшем будем считать, что «логическое» многоуровневое представление системы существует.

Обозначения используемых в системе переменных должны быть приведены в соответствие с глобальной идентификацией. Это может быть сделано, например, с помощью соответствующих индексов. Чтобы не перегружать формальную модель чрезмерной индексацией, в дальнейшем (если это не нарушает однозначности) будем использовать локальные обозначения переменных, принятых в первой модели. Это возможно, поскольку в большей части изменения переменных производятся только в *локальных областях* и чаще всего в пределах экземпляра ФБ. Локальная область изменения переменных предварительно будет оговорена.

Проблем с использованием обозначений связей элементов, локализованных в пределах модуля ФБ, в структуре Крипке нет. При использовании связей, имеющих отношение к интерфейсным элементам – формальным параметрам (рис. 3.11, 3.12), будем считать, что построены отображения, связывающие их с фактическими параметрами родительского ФБ. Таким образом, неявно предполагается, что вместо формального параметра используется соответствующий ему фактический параметр.

Для упрощения построения модели можно принять следующие *предположения*: 1) каждое *EC*-состояние имеет хотя бы одну *EC*-акцию. Если в действительности такого нет, то вводится фиктивная *EC*-акция («пустая» *EC*-акция), которая не выполняет никаких действий; 2) каждая *EC*-акция имеет алгоритм. Если на самом деле такого не наблюдается, то вводится фиктивный алгоритм, содержащий один шаг, единственным действием которого является сброс счетчика шагов в ноль.

При рассмотрении переходов, относящихся к функционированию ФБ, будем ограничиваться рассмотрением только тех переменных, которые описывают динамику функционирования одного экземпляра ФБ. Рассмотрим каждый из переходов отдельно.

Переходы типа «Срабатывание *EC*-перехода» бывают нескольких модификаций в зависимости от следующих условий: а) является ли соответствующий *EC*-переход нагруженным событием или нет; б) содержит ли целевое состояние *EC*-акции или нет. В случае триггерных *EC*-переходов, для которых целевое *EC*-состояние имеет *EC*-акции, соответствующее подмножество переходов из T_{FT}^{B} представляется следующим множеством правил:

$$\{T_{FT}^{B,1}[i,k,j]: Z_S(S) = s_1 \wedge \bigvee_{(q_i, ei_k, q_j) \in ECTran} (Z_Q(Q) =$$

$$= q_i \wedge EnabledECTran(q_i, ei_k, q_j, Z_{EI}, Z_V) \wedge$$

$$\wedge \bigwedge_{\substack{(q_i, y, q_m) \in ECTran, \\ (q_i, y, q_m) \prec (q_i, ei_k, q_j)}} \overline{EnabledECTran(q_i, y, q_m, Z_{EI}, Z_V)}) \Rightarrow$$

$$\Rightarrow Z_Q(Q) \leftarrow q_j; Z_S(S) \leftarrow s_2;$$

$$\Xi_{ei_j \in EI} Z_{EI}(ei_j) \leftarrow false; Z_{NA}(NA) \leftarrow 1; Z_{NI}(NI) \leftarrow 1 \mid (q_j, ei_k, q_j) \in$$

$$\in ECTran, q_j \in Q^A, ei_k \in EI\},$$

где $Q^A \subseteq DomQ$ – множество *EC*-состояний, имеющих прикрепленные *EC*-акции.

Приведенный переход включает следующие одновременные действия: 1) изменение *EC*-состояния с q_i на q_j; 2) изменение *OSM*-состояния с s_1 на s_2; 3) установку счетчика *EC*-акций *NA* в единицу; 4) установку счетчика шагов алгоритмов *NI* в единицу; 5) сброс всех входных событийных переменных.

Для нетриггерных *EC*-переходов, целевые *EC*-состояния которых не имеют *EC*-акций, множество правил, представляющих переходы «Срабатывание *EC*-перехода», следующее:

$$\{T_{FT}^{B,2}[i,j]: Z_S(S) = s_1 \wedge \bigvee_{(q_i, \varepsilon, q_j) \in ECTran} (Z_Q(Q) =$$

$$= q_i \wedge EnabledECTran(q_i, \varepsilon, q_j, Z_{EI}, Z_V) \wedge$$

$$\wedge \bigwedge_{\substack{(q_i, y, q_m) \in ECTran, \\ (q_i, \varepsilon, q_j) \prec (q_i, y, q_m)}} \overline{EnabledECTran(q_i, y, q_m, Z_{EI}, Z_V)}) \Rightarrow Z_Q(Q) \leftarrow q_j;$$

$$Z_S(S) \leftarrow s_2;$$

$$Z_{NA}(NA) \leftarrow 0; Z_{NI}(NI) \leftarrow 0 \,|\, (q_j, \varepsilon, q_j) \in ECTran, q_j \in Q \setminus Q^A\}.$$

Как видно, данное правило включает меньшее число изменений переменных, чем предыдущее. Для краткости изложения другие разновидности правил типа «Срабатывание *EC*-перехода» здесь не приводятся.

Как было отмечено выше, существует несколько стратегий (принципов) съема данных. Для определенности выберем синхронный съем данных и определим соответствующие правила. Для рассмотрения перехода типа «Синхронный съем данных» введем функцию (инъективное отображение) $f_{EV} : 2^{EI} \to 2^{VI}$, ставящую в соответствие множеству входных событий множество входных переменных, для которых они производят съем данных. Данная функция строится следующим образом:

$$(EI_i, VI_j) \in f_{EV}, \text{åñëè } VI_j = \bigcup_{ei_k \in EI_i} \{vi_m \,|\, (ei_k, vi_m) \in IW\}.$$

Иными словами, множество VI_j образуется только из тех входных переменных, которые связаны *WITH*-связями с входными событиями из множества EI_i.

Число переходов типа «Синхронный съем данных» будет равно $|\,\mathrm{Pr}_2\, f_{EV}\,|$, где $\mathrm{Pr}_2 f_{EV}$ обозначена вторая проекция отношения f_{EV}. Множество правил, определяющее все переходы типа «Синхронный съем данных», представлено ниже:

$$\{T_{SD}^{B,1}[j] : Z_\alpha(\alpha) \wedge Z_S(S) = s_0 \wedge \bigvee_{(EI_k, VI_j) \in f_{EV}} \Big(\bigwedge_{ei_m \in EI_k} Z_{EI}(ei_m) \wedge$$

$$\bigwedge_{ei_n \in EI \setminus EI_k} \overline{Z_{EI}(ei_n)} \Big) \Rightarrow \Xi_{vi_i \in VI^j} (Z_{VI}(vi_i) \leftarrow Z_{VOB}(pre_{VI}(vi_i)));$$

$$Z_S(S) \leftarrow s_1 \,|\, VI_j \in \mathrm{Pr}_2\, f_{EV}, VI_j \neq \varnothing\},$$

где знаком Ξ обозначена операция параллельного (синхронного) выполнения операций, стоящих под этим знаком;

$pre_{VI} : \bigcup_{M^i \in SM_B \cup SM_C} VI^i \to \bigcup_{M^j \in SM_B} VOB^j$ – глобальная функция, позволяющая определить для каждой входной переменной экземпляра базисного ФБ (МБФБ) соответствующий ей буфер данных, где VI^i – множество входных переменных *i*-го МБФБ; VOB^j – множество выходных буферов *j*-го экземпляра базисного ФБ (МБФБ).

При выполнении данного правила выполняются следующие одновременные действия: 1) производится запись значений из соответствующих буферов данных во входные переменные (из множества VI_j); 2) изменяется *OSM*-состояние с s_0 на s_1.

Для реализации перехода «Выполнение шага алгоритма» введем предикат, определяющий условие разрешенности шага алгоритма:

$$CondStep : Dom(Q) \times Dom(NA) \times Dom(NI) \times [Z_V] \to \{true, false\}.$$

Данный предикат похож на предикаты *CondNI*, $CondVO_m$ и $CondVV_m$, введенные выше, но ориентирован на весь шаг, а не на отдельные переменные. Также будем считать, что функция *nextSt* выдает значение «0», если выполняется последний шаг алгоритма.

Переходы «Выполнение шага алгоритма» могут быть представлены в виде следующего множества правил:

$$\{T_{EX}^{B,1}[i,j,k]: Z_S(S) = s_2 \wedge Z_Q(Q) = q_i \wedge Z_{NA}(NA) = j \wedge Z_{NI}(NI) = k \wedge$$

$$\wedge CondStep(q_i, j, k, Z_V) \Rightarrow \underset{vo_m \in VO^{q_i,j,k}}{\Xi} (Z_{VO}(vo_m) \leftarrow NewValVO_m(q_i, j, k, Z_V));$$

$$\underset{vv_m \in VV^{q_i,j,k}}{\Xi} (Z_{VV}(vv_m) \leftarrow NewValVV_m(q_i, j, k, Z_V));$$

$$Z_{NI}(NI) \leftarrow nextSt(q_i, j, k, Z_V) \mid q_i \in Q^A, j \in NA^{q_i}, k \in NI^{q_i,j}\},$$

где $VO^{q_i,j,k}$ и $VV^{q_i,j,k}$ – множество выходных и внутренних переменных базисного ФБ, изменяемых в *k*-м шаге алгоритма *j*-й *EC*-акции в *EC*-состоянии q_i соответственно; Q^A – множество *EC*-состояний, к которым прикреплены *EC*-акции; NA^{q_i} – множество номеров *EC*-акций в *EC*-состоянии q_i, в которых производится изменение переменных; $NI^{q_i,j}$ – множество номеров шагов в *j*-й *EC*-акции в *EC*-состоянии q_i, в которых производится изменение переменных.

При выполнении данного правила производятся следующие одновременные действия: 1) изменение внутренних и выходных переменных, входящих в шаг согласно соответствующим выражениям, стоящим в правых частях операторов присваивания; 2) изменение счетчика шагов согласно управляющей структуре алгоритма.

Переход «Завершение выполнения *EC*-акции» может иметь несколько разновидностей в зависимости от того, связан ли с *EC*-акцией выходной сигнал и если да, то инициирует ли этот выходной сигнал выдачу выходных данных. Правило, представляющее пере-

ход типа «Завершение выполнения *EC*-акции (без выдачи выходных сигналов и данных)» приведено ниже:

$$T_{CA}^{B,1}: Z_S(S) = s_2 \wedge Z_{NI}(NI) =$$
$$= 0 \wedge \bigvee_{q_i \in Q^{noEO}} \bigvee_{j \in NA^{q_i,noEO}} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) = j) \Rightarrow$$
$$\Rightarrow Z_{NA}(NA) \leftarrow Z_{NA}(NA) + 1; Z_{NI}(NI) \leftarrow 1,$$

где $Q^{noEO} \subseteq Q-$ множество *EC*-состояний, хотя бы в одной *EC*-акции которых не выдается выходной сигнал; $NA^{q_i,noEO}$ – множество номеров *EC*-акций *EC*-состояния q_i, в которых не выдается выходной сигнал.

При выполнении данного правила выполняется счетчик *EC*-акций увеличивается на единицу, а счетчик шагов устанавливается равным единице.

Множество правил, представляющих переходы типа «Завершение *EC*-акций (с выдачей выходного сигнала и с выдачей данных)» представлено ниже:

$$\{T_{CA}^{B,2}[k]: putoutEO_k \Rightarrow Z_{NA}(NA) \leftarrow Z_{NA}(NA) + 1; Z_{EO}(eo_k) \leftarrow true;$$
$$\Xi_{(eo_k, vo_m) \in OW} Z_{VOB}(vob_m) \leftarrow Z_{VO}(vo_m) \mid eo_k \in EO\}.$$

Выполняемые действия в переходах данного типа: 1) инкремент счетчика *EC*-акций; 2) установка выходной событийной переменной, чем и определяется факт выдачи выходного сигнала; 3) выдача выходных данных наружу блока, что выражается в записи значений выходных переменных, связанных *WITH*-связями с выходным событием, в соответствующие буфера данных.

Переход «Завершение выполнения *EC*-состояния» выражается простым правилом

$$T_{CS}^{B,1}: Z_S(S) = s_2 \wedge \bigvee_{q_i \in DomQ} (Z_Q(Q) = q_i \wedge Z_{NA}(NA) > N_A^{q_i}) \Rightarrow$$
$$\Rightarrow Z_S(S) \leftarrow s_1; Z_{NA}(NA) \leftarrow 1; Z_{NI}(NI) \leftarrow 1.$$

Данный переход активизируется, когда в состоянии выполнения ФБ (*OSM*-состояние s_2) завершено выполнение всех *EC*-акций, прикрепленных к *EC*-состоянию (определяется путем сравнения счетчика *EC*-акций с числом *EC*-акций в каждом конкретном *EC*-состоянии). Действия, выполняемые одновременно при активизации данного перехода: 1) установка *OSM*-состояния s_1; 2) установка счетчиков *EC*-акций и шагов алгоритмов в единицу.

Переход «Завершение выполнения базисного ФБ» выражается следующим правилом:

$$T_{CF}^{B,1} : Z_S(S) = s_1 \wedge AbsentsEnabledECTran \Rightarrow$$
$$\Rightarrow Z_S(S) \leftarrow s_0; Z_\alpha(\alpha) \leftarrow false; Z_\beta(\beta) \leftarrow true.$$

Данный переход активизируется, когда в *OSM*-состоянии s_1 нет разрешенных *EC*-переходов. Действия, выполняемые одновременно при активизации данного перехода: 1) переход базисного ФБ в состоянии «Свободен»; 2) сброс переменной запуска ФБ; 3) установка переменной окончания выполнения ФБ.

Как отмечалось выше, при циклической модели выполнения диспетчер запускает «свободный» базисный ФБ независимо от наличия на его входах сигналов и от его готовности или неготовности обработать эти входные сигналы. Поэтому в данном случае возможна ситуация так называемого «пустого» выполнения базисного ФБ, когда никаких действий базисным ФБ фактически не выполняется. Переход «Пустой запуск базисного ФБ» представляется правилом:

$$T_{ES}^{B,1} : Z_\alpha(\alpha) \wedge Z_S(S) = s_0 \wedge \bigwedge_{ei_k \in EI} \overline{selectEI_k} \Rightarrow Z_\alpha(\alpha) \leftarrow false; Z_\beta(\beta) \leftarrow true.$$

Данный переход активизируется, когда в «свободном» состоянии базисного ФБ приходит сигнал его запуска, но при этом ни один из его входных сигналов не выбран. Действия, выполняемые одновременно при активизации данного перехода: 1) сброс переменной запуска ФБ; 2) установка переменной окончания выполнения ФБ.

Переходы типа «Синхронный съем данных в составном ФБ», описываются следующим множеством правил:

$$T_{TC}^{C}[j] : Z_\alpha(\alpha) \wedge \bigwedge_{ei_m \in EI_j} Z_{EI}(ei_m) \wedge \bigwedge_{ei_n \in EI \setminus EI_j} \overline{Z_{EI}(ei_n)} \Rightarrow$$

$$\Rightarrow \underset{\substack{(EI_j, VI_k) \in f_{EV}, \\ vi_i \in VI_k}}{\Xi} (Z_{VI}(vi_i) \leftarrow Z_{VOB}(pre_{VI}(vi_i)));$$

$$\underset{ei_m \in EI_j}{\Xi} \underset{\substack{(ei_m, ei_k^x) \in \\ \in EvConn}}{\Xi} Z_{EI^x}(ei_k^x) \leftarrow true;$$

$$\underset{ei_m \in EI_j}{\Xi} Z_{EI}(ei_m) \leftarrow false \mid EI_j \in 2^{EI}, EI_j \neq \varnothing\}.$$

Число переходов данного типа равно числу подмножеств, образованных из множества событийных входов (исключая пустое мно-

жество). Переход типа T_{TC}^{C} является разрешенным (по отношению к j-му подмножеству из 2^{EI}), если все событийные входы из этого подмножества «маркированы» сигналами, в то время как на остальных входах ФБ сигналы отсутствуют. При выполнении этого перехода выполняются следующие действия: 1) устанавливаются в значение «Истина» все целевые событийные переменные, которые связаны с маркированными событийными входами; 2) осуществляется съем данных по тем информационным линиям, которые связаны *WITH*-ассоциациями с маркированными событийными входами; 3) сбрасываются все входные события ФБ.

Переходы типа «Передача и выдача сигналов составным ФБ» могут быть представлены следующим множеством обобщенных правил:

$$\{T_{TO}^{C,1}[x,k]: Z_{EO^x}(eo_k^x) \Rightarrow Z_{EO^x}(eo_k^x) \leftarrow false;$$

$$\underset{(eo_k^x, ei_m^y) \in EvConn}{\Xi} Z_{EI^y}(ei_m^y) \leftarrow true;$$

$$\underset{(eo_k^x, eo_j) \in EvConn}{\Xi} (Z_{EO}(eo_j) \leftarrow true;$$

$$\underset{(eo_j, vob_m) \in OW}{\Xi} Z_{VOB}(vob_m) \leftarrow Z_{VO}(repr_{VO}(vob_m)))|$$

$$| x \in \overline{1, N_{FB}}, eo_k^x \in EO^x\}.$$

Переход данного типа обрабатывает один из активных источников сигнала, коим является событийный выход компонентного ФБ. При срабатывании данного правила выполняются следующие действия: 1) передаются сигналы на целевые событийные входы компонентных ФБ и на целевые событийные выходы оболочки; 2) производится выдача данных по тем информационным линиям, которые связаны *WITH*-ассоциациями с целевыми событийными выходами; 3) сбрасывается источник сигнала как отработанный. Из приведенного выше правила могут быть получены более частные правила в зависимости от конкретной топологии соединений. В частности, в правиле может отсутствовать съем данных.

Как было отмечено ранее, существуют два подвида диспетчера для циклической модели: главный диспетчер и диспетчер промежуточного уровня. Разница между ними состоит в том, что главный диспетчер функционирует автономно и циклически, а диспетчер промежуточного уровня – по запросу. Тем не менее оба диспетчера являются очень похожими, поэтому рассмотрим переходы диспет-

чера промежуточного уровня. Переходы типа «Запуск компонентного ФБ» могут быть представлены следующими правилами:

$$T_{SF}^{D,1} : Z_{\alpha}(\alpha) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\alpha_1}(\alpha_1) \leftarrow true;$$

$$\{T_{SF}^{D,2}[i] : Z_{\beta_i}(\beta_i) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\alpha_{i+1}}(\alpha_{i+1}) \leftarrow true;$$

$$Z_{\beta_i}(\beta_i) \leftarrow false \,|\, i = \overline{1,(N_{FB}-1)}\}.$$

Первое правило предназначено для запуска первого компонентного ФБ, а второе – для запуска последующих ФБ. Второе правило является разрешенным, если закончил выполнение предыдущий компонентный ФБ и при этом все передачи сигналов в самом составном ФБ закончены. Определяемые правилом действия сводятся к запуску последующего компонентного ФБ, а также сбросу переменной окончания выполнения текущего компонентного ФБ.

Переход «Завершение выполнения составного ФБ» совершается тогда, когда закончил выполнение последний компонентный ФБ в составном ФБ. Правило для данного перехода следующее:

$$T_{DC} : Z_{\beta_{N_{FB}}}(\beta_{N_{FB}}) \wedge Z_{\omega}(\omega) \Rightarrow Z_{\beta}(\beta) \leftarrow true; Z_{\alpha}(\alpha) \leftarrow false.$$

Идея и основные положения представления операционной семантики ФБ IEC 61499 в виде структуры Крипке были представлены в работе [18].

4. Проверка моделей систем функциональных блоков

Обоснование выбора метода и средства верификации

При выборе методов и средств верификации ФБ следует руководствоваться:

1) основополагающей моделью поведения, описывающей как собственно системы ФБ, так и объекты управления, поскольку зачастую верифицируется замкнутая система; 2) возможностью описывать требования (свойства), предъявляемые к системе ФБ и замкнутой системе в целом; 3) сложностью и ресурсозатратностью метода верификации; 4) наличием инструментальных средств поддержки.

В качестве метода верификации ФБ в нашем случае выбираем метод *Model checking* исходя из следующих соображений: 1) ФБ имеет (или должен иметь) конечное число состояний, что является необходимым условием применения метода; 2) данный метод хорошо зарекомендовал себя на практике как для верификации программ, так и аппаратного обеспечения. Следует отметить, что ФБ обладают свойствами как того, так и другого; 3) имеются инструментальные системы поддержки данного метода.

В качестве инструментальной системы верификации выбираем систему *SMV*. Аргументами в пользу такого выбора являются: 1) адекватность системы *SMV* основополагающим моделям поведения, какими в нашем случае являются две модели формальной семантики ФБ – на основе машин абстрактных состояний (МАС) (так называемые ФМОСФБ) и на основе структуры Крипке. Интуитивно МАС и *SMV* схожи, поскольку обе представляют системы переходов. Только в *SMV* состояние формируется как совокупность значений переменных состояния, а в МАС в качестве состояния используются структуры (алгебры). Переход в *SMV* представляется как изменение переменных состояния, а МАС – как изменение значений функции. Поскольку в модели 1 семантики ФБ (ФМОСФБ) изменяются, как правило, только функции значений (что, по сути, эквивалентно изменению значений переменных), то это делает переход от ФМОСФБ на язык *SMV* довольно простым. Следует отметить, что в работе [254] представлен общий метод преобразования МАС в код *SMV*. Модель 2 семантики в виде структуры Крипке идеально подходит для метода *Model checking* и отображается 1:1 в код *SMV*; 2) наличие в системе *SMV* средств для спецификации свойств сис-

темы в виде формул временных логик *LTL* и *CTL*; 3) хорошие характеристики системы *SMV* (*NuSMV*) как по быстродействию, так и по потребляемой памяти; 4) поддержка композиционных методов верификации [106].

Особенности использования *SMV* в технологической цепочке верификации систем ФБ

Можно выделить два подхода к разработке моделей в системе *SMV*. Первый подход предполагает использование концепции модулей. Модульная модель на *SMV* интерпретируется с использованием смешанной синхронно-асинхронной семантики. Все действия по изменению переменных состояния внутри модуля *SMV* выполняются по шагам синхронно с использованием оператора *next*, в то время как модули один относительно другого могут выполняться как синхронно, так и асинхронно. Модульную *SMV*-модель можно представить в виде структуры Крипке, для «сборки» которой использовалась синхронная и асинхронная композиция структур Крипке, представляющих модули. Данный подход является «дружественным» для пользователя. Он поддерживает иерархическое проектирование и дает возможность один к одному отобразить иерархию системы ФБ на иерархию модулей *SMV*. Кроме того, он дает возможность использования композиционных методов верификации, поддерживаемых в системе *SMV*. Недостатком подхода является невысокое быстродействие системы верификации в данном режиме.

Второй подход ориентирован на явное определение всех возможных переходов в системе с использованием предложений *TRANS* и *INIT*. Данный подход является значительно более быстрым в вычислении свойств модели и не создает проблем с использованием свойства «справедливости» (*fairness*). Во многих исследовательских работах используется второй подход, например [107, 110]. Но, как отмечено в руководстве по языку *SMV* [177], этим подходом надо пользоваться очень осторожно, поскольку при неполном описании переходов система *SMV* может выдать непредсказуемые результаты. Существенным недостатком данного подхода является использование концепции глобального состояния и отсутствие поддержки иерархического проектирования. Поэтому предварительно явно или неявно модель иерархической системы должна быть приведена к «плоской» структуре (по крайней мере, на уровне имен). Отсутствие понятия модулей и механизма передачи параметров осложняет задание соответствия между переменными и их представителями. Ручное построение модели сложной системы ФБ в соответ-

ствии с данным подходом является проблематичным, поэтому требуется построение *автоматического транслятора*.

Рассмотрим иерархию моделей, используемых при описании систем ФБ, для уточнения терминологии, используемой в данном разделе. Формальную нотацию для представления операционной семантики систем ФБ будем считать некоторой формальной *мета-метамоделью*. Определение формальной семантики ФБ с использованием подобной *метаметамодели* можно считать формальной *метамоделью* систем ФБ. А описание конкретной системы ФБ на основе этой метамодели будем называть формальной *моделью* системы ФБ. Преобразование формальной модели в программу на каком-либо языке программирования, моделирования или описания аппаратуры будем называть *реализацией*.

Технология моделирования систем ФБ на *SMV* включает следующие этапы: 1) построение для исследуемой системы ФБ формальной модели с использованием одной из двух предложенных в разделе 3 формальных метамоделей; 2) реализацию формальной модели системы ФБ на входном языке *SMV*. Ввиду близости формальной модели и модели *SMV* этап 1 может быть в принципе опущен. Однако это возможно только в случае хорошего овладения формальной метамоделью. Формальная метамодель может быть положена в основу построения автоматического транслятора описания систем ФБ в код *SMV*.

Ниже рассматриваются методы преобразования исходного описания системы ФБ в код *SMV* на основе двух ранее предложенных формальных метамоделей. При иллюстрации подходов явная формальная модель системы ФБ не строится, однако даются ссылки на используемые элементы формальной метамодели.

4.1. Подход на основе отображения машин абстрактных состояний в модули *SMV* (подход 1)

Как было отмечено выше, для преобразования ФМОСФБ в модель *SMV* можно воспользоваться результатами работы [254], в которой описан метод трансформации МАС в *SMV*.

Для отображения формальной модели на основе ФМОСФБ в модель *SMV* выберем первый технологический подход (на основе модулей *SMV*) как наиболее адекватный. В этом случае семантический разрыв между ФМОСФБ и моделью *SMV* получается минимальным. Каждому модулю ФМОСФБ (далее просто формальному

модулю) можно поставить в соответствие модуль *SMV*. Как модули ФМОСФБ, так и модули *SMV* работают друг по отношению к другу асинхронно, в то время как правила обновления функций модуля ФМОСФБ и операторы *next* модуля *SMV* выполняются синхронно.

Объявление модуля в *SMV* имеет следующий синтаксис:

MODULE <имя модуля>(<список формальных параметров>)

Главный модуль *SMV* определяется как

MODULE main [(<список формальных параметров>)]

Главный модуль *SMV* используется для моделирования системы ФБ верхнего уровня. Как правило, это приложение IEC 61499, но таким же образом можно промоделировать базисные и составные ФБ. Следует, однако, при этом учитывать, что при построении пространства состояний будут использованы всевозможные значения указанных параметров модуля.

Переменным формального модуля соответствуют формальные параметры или собственные переменные модуля *SMV*. Раскладка переменных может быть произведена на основе рис. 3.11 и 3.12 из подраздела 3.10. В этом случае реальным переменным, обозначенным сплошными окружностями, соответствуют собственные переменные модуля *SMV*, а представителям, обозначенным пунктирными окружностями, – формальные параметры модуля *SMV*.

Из всех типов переменных, определенных для ФБ IEC 61499, в *SMV* можно использовать только логический и целочисленный типы, что бывает вполне достаточно для большинства случаев. Событийные переменные описываются с использованием логического типа. Целочисленный тип в *SMV* определяется диапазоном, например: –99...+99. Следует заметить, что в *SMV* не поддерживается вещественный тип данных, что, возможно, связано с очень большим пространством состояний при его использовании, но данный тип и работа с ним при необходимости могут быть эмулированы на основе имеющихся типов данных. Переменные модуля описываются с использованием *VAR*-предложения, имеющего синтаксис:

VAR <список переменных>: <тип данных>;

Инициализация переменных определяется оператором *INIT*:

INIT(<имя переменной>):= <выражение>;

Каждому правилу (или набору правил) модуля формальной модели по изменению переменной в модуле *SMV* соответствует оператор *next* с оператором *case* в правой части оператора присваивания:

next(<имя переменной>):= *case*

<условие 1 изменения переменной>: <выражение 1 для нового значения>;

…

<условие N изменения переменной>: <выражение N для нового значения>;

esac;

Вычисление данной конструкции производится следующим образом. В текущем такте последовательно вычисляются условия изменения переменной, указанные в операторе *case*. Как только окажется истинным какое-либо условие, в соответствии с указанным выражением вычисляется новое значение переменной, которое будет присвоено переменной в следующем такте. А выполнение оператора *case* прекращается, и последующие условия не вычисляются. Если при определенных условиях необходимо сохранение значения переменной, то в *case*-операторе следует использовать конструкцию:

<условие сохранения переменной>: <имя переменной>;

Обычная данная конструкция стоит в конце *case*-оператора и имеет вид

1: <имя переменной>;

что соответствует тому, что если предыдущие условия не выполняются, то значение переменной сохраняется.

Каждому компонентному ФБ формальной модели в *SMV* ставится в соответствие описание (вызов) соответствующего *SMV*-модуля в объемлющем *SMV*-модуле:

VAR <имя экземпляра модуля ФБ>: *process* <имя типа модуля ФБ>(<параметры экземпляра модуля ФБ>);

Следует заметить, что ключевое слово *process* в данном случае описывает выполнение порожденного экземпляра модуля как асинхронного процесса. При отсутствии данного ключевого слова порожденный экземпляр модуля будет выполняться синхронно по отношению к вызывающему модулю.

При использовании часто повторяющихся логических условий в операторе *case* могут использоваться отдельно определенные с помощью *DEFINE*-предложения логические выражения. Например, таким образом целесообразно определить условия *EC*-переходов и сторожевые условия, условия наличия/отсутствия сигналов на событийных входах, условие завершенности передач в составных ФБ и т.п. Синтаксис *DEFINE*-предложения для определения логических условий следующий:

DEFINE {<имя логического условия>:= <логическое выражение>;}…

В табл. 4.1 приведено соответствие элементов формального модуля и модуля *SMV*.

Таблица 4.1

Соответствие элементов формального модуля элементам модуля *SMV*

ФМОСФБ	Модель *SMV*
(Формальный) модуль	Модуль *SMV*
Простое правило по изменению функции значения переменной (в виде одного продукционного правила)	Оператор *next* с оператором присваивания
Составное правило по изменению функции значения переменной (в виде группы продукционных правил)	Оператор *next* с оператором *case*
(Реальная) переменная	Собственная переменная модуля *SMV*
Переменная-представитель	Формальный параметр модуля *SMV*
Компонентный ФБ	Переменная, представляющая экземпляр модуля указанного типа с описателем *process*

Для формулировки и доказательства свойств системы ФБ на основе временной логики *CTL* используется *SPEC*-предложение:

SPEC <формула временной логики CTL>

Правила построения формул временной логики можно найти в руководстве по языку *SMV* [177].

Следует обратить внимание на особенности использования конструкции *FAIRNESS* вообще и *FAIRNESS running* в частности. Системная переменная *running* показывает активность модуля *SMV* в каждом такте. По идее, использование данной конструкции в какомлибо модуле *SMV* «заставляет» этот модуль выполняться бесконечно часто. Таким образом, верификатор ограничивается рассмотрением только тех путей, где модуль выполняется бесконечно часто.

Использовать или не использовать конструкцию *FAIRNESS running* – зависит от класса решаемых задач и типа вычисляемых *CTL*-формул. При использовании в системе конструкции *FAIRNESS running* проверяемая *CTL*-формула, содержащая временной оператор *EF*, будет истинна, если не будет тупикового пути, на протяжении которого эта формула будет ложна (даже в том случае, если существует другой путь, в котором эта формула является истинной). Иными словами, если мы знаем, что формула *EF*(*p*) очевидно ис-

тинна, но при этом существует тупиковый путь, на протяжении которого формула *p* является ложной, то при использовании конструкции *FAIRNESS running* формула *EF*(*p*) превращается в ложную, поскольку при использовании этой конструкции производится проверка только на бесконечных путях.

Следует отметить, что в ряде случаев использование конструкции *FAIRNESS running* в системе *Cadence SMV* приводит к очень длительному процессу доказательства по сравнению с вариантом, где конструкция *FAIRNESS running* отсутствует. Например, при верификации системы ФБ из двух АЛУ в циклической модели выполнения без конструкции *FAIRNESS* для вычислений требовалось 2–3 мин, а с использованием *FAIRNESS* – минут 40–60. Поэтому при решении только задачи достижимости с использованием спецификации *SPEC EF* конструкцию *FAIRNESS running* лучше не использовать. Это будет действительным, если исключен системный тупик (см. выше).

При доказательстве свойства типа *SPEC AG*(...->*AF*) и *SPEC* !*EF* конструкцию *FAIRNESS running* рекомендуется использовать. Без нее может быть (неверно) вычислено *FALSE* и выдана трасса контрпримера, но при этом будет видно, что один из модулей бесконечно «циклится».

Для того, чтобы (многократно) повысить скорость вычислений, можно использовать *TRANS*-предложения. С использованием конструкции *TRANS* можно явно представить все переходы в формальной модели. При этом переходы в описании разделяются знаком |. Для каждого перехода описываются условие совершения перехода, а также то, как связано предыдущее и последующее значение каждой переменной модели. Недостатком данного подхода является громоздкость *TRANS*-предложений, что делает его использование возможным только при использовании специального автоматического транслятора.

4.2. Подход на основе использования структуры Крипке и предложений *TRANS* в *SMV* (подход 2)

Для представления формального описания системы ФБ, заданной в виде структуры Крипке (в соответствии с метамоделью 2), на языке *SMV* (подход 2) не требуется построения никаких «семантических мостов». Каждому переходу формальной модели соответствует описание одного перехода с помощью *TRAN*-предложения.

Однако при этом из-за использования концепции глобального состояния и отсутствия модульности возникают некоторые проблемы «конструктивного» характера. Как было отмечено выше, данный подход требует развертывания системы, а значит – использования глобальных имен. При этом структурированное пространство имен, состоящее из локальных пространств имен для модулей, преобразуется в общее (линейное) пространство имен всей системы. Могут быть использованы различные способы образования глобальных имен. Один из вполне приемлемых вариантов состоит в использовании иерархических имен, отражающих путь в дереве иерархий от корня до требуемого экземпляра ФБ. В этом случае имя экземпляра ФБ определяется как

<глобальное имя экземпляра ФБ>::=

{<имя компонентного ФБ промежуточного уровня>_}…< имя компонентного ФБ>

Причем первым в таком иерархическом имени идет имя компонентного ФБ самого высокого уровня иерархии, а последним имя требуемого компонентного ФБ. Пример иерархического имени: *fa*2*_fb*7*_fc*3. Имена переменных в глобальной модели могут образоваться следующим образом:

<глобальное имя переменной ФБ>::=

<локальное имя переменной ФБ>_<глобальное имя экземпляра ФБ>

Пример глобального имени переменной: *ei*1*_ fa*2*_fb*7*_fc*3. Как можно заметить, глобальное имя экземпляра ФБ используется в качестве суффикса глобального имени переменной.

Идентификаторы (имена) модулей должны определяться априорно, до начала моделирования. В системах, в которых используется типизация (например, в *SMV*) при именовании могут возникнуть дополнительные проблемы. Это происходит из-за того, что развертывание системы экземпляров из системы типов производится автоматически, причем может быть несколько экземпляров одного и того же типа. Для решения проблемы предлагается *динамическое* вычисление идентификаторов модулей. Каждому экземпляру модуля ФБ передается идентификатор родительского модуля и локальный идентификатор модуля ФБ в родительском модуле. При развертывании системы каждый экземпляр модуля вычисляет собственный идентификатор путем конкатенации идентификатора родительского модуля и локального идентификатора. На рис. 4.1 приведен пример вычисления идентификаторов модулей в трехуровневой

системе. Рядом с каждым модулем поставлен его глобальный и локальный идентификатор. Для представления глобальных идентификаторов модулей в данном случае используется точечная нотация.

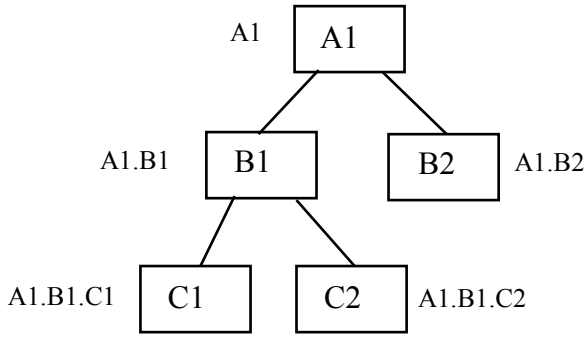


Рис. 4.1. Пример вычисления идентификаторов модулей

Структура *SMV*-модели при использовании *TRANS*-предложений может быть следующей:

MODULE main

<раздел объявления переменных>

<раздел начальных значений переменных>

<раздел описания переходов>

<раздел предопределенных условий>

<раздел спецификаций свойств>

Как можно увидеть, модель *SMV* состоит только из одного главного модуля. Объявления переменных аналогичны объявлениям в предшествующей *SMV*-модели.

Начальные значения переменных задаются в *INIT*-предложении в виде конъюнкции. Терм такой логической формулы имеет вид

<имя переменной>=<начальное значение переменной>

Раздел описания переходов в нашем случае состоит из одного предложения *TRAN*, которое содержит описание всех переходов системы. Данное описание фактически по форме представляет формулу, содержащую конъюнкты, разделенные знаком дизъюнкции «|». Каждый конъюнкт описывает отдельный переход. Все термы конъюнкта связаны между собой символом конъюнкции «&». Функционально каждый конъюнкт делится на две части: условную часть («голову»), которая определяет условие «разрешенности» перехода, и исполнительную часть («тело»), содержащую действия над переменными, которые будут выполнены одновременно при «срабатывании» перехода. Структурно же для удобства будем разбивать переход на три части: условную, исполнительную и сохра-

няющую часть (в виде условий сохранения), речь о последней пойдет ниже. В принципе «условная часть» может представлять любую логическую формулу над переменными состояния, в которой использованы логические операции конъюнкции, дизъюнкции и отрицания. Исполнительная часть состоит из выражений вида *next*(<*имя переменной*>)=<*выражение*>, связанных операциями конъюнкции «&». Динамика перехода заключается в том, что непосредственно после выполнения перехода переменные, указанные после слова *next*, примут новые значения, вычисленные в соответствии с выражением, стоящим справа от знака «=». При необходимости оставить переменную без изменений следует использовать терм вида: *next*(<*имя переменной*>)=< *имя переменной*>. В одном глобальном состоянии может быть несколько разрешенных переходов, в чем и проявляется недетерминизм моделируемой системы. Система *SMV* исследует все возможные переходы. Следует обратить внимание на возможность зацикливания на выполнении одного и того же правила, если это правило не изменяет переменные, входящие в свой антецедент.

В разделе предопределенных условий с использованием *DEFINE*-предложений целесообразно определить многократно используемые условия (подформулы) модели. В целом их можно разделить на два класса: 1) условия, определяющие логику функционирования ФБ; 2) условия сохранения значений групп переменных. К первому классу условий относятся, например, следующие:

GuardCond – сторожевые условия *EC*-переходов;

EnabledECTran – условия разрешенности *EC*-переходов;

ExistsEnabledECTran – условие существования разрешенных *EC*-переходов;

AbsentsEnabledECTran – условие отсутствия разрешенных *EC*-переходов;

omega – условие окончания передач в составном ФБ.

Все эти условия использовались в формальной модели и имели практически те же имена (за исключением последнего условия, обозначенного буквой ω). Следует отметить, что данные условия должны быть определены для каждого экземпляра ФБ. При этом при образовании глобального имени условия к локальному имени может быть добавлено иерархическое имя модуля, например, *ExistsEnabledECTran_fa*2*_fb*7*_fc*3.

Ко второму классу условий относятся условия сохранения значений для определенных групп переменных. Необходимость данных условий обусловлена тем, что правила построения глобальных

переходов на языке *SMV* требуют определения значений для всех переменных, входящих в глобальное состояние. А поскольку при выполнении перехода фактически изменяется только очень ограниченная часть переменных, а остальная часть переменных остается неизменной, то имеет смысл вынести эту незначащую неизменяемую часть в отдельный раздел (чтобы она не затемняла смысл основной части).

На основе анализа частоты «неиспользования» в переходах определенных групп переменных решено выделить следующие (типы) условий сохранения значений:

u_inner_obj – сохранение внутренних объектов базисного ФБ;

u_ext_obj – сохранение внешних объектов базисного ФБ;

u_obj: = u_inner_ob & u_ext_obj – сохранение всех объектов базисного ФБ;

u_input_var – сохранение входных переменных ФБ;

u_int_out_var – сохранение выходных и внутренних переменных базисного ФБ;

u_buf – сохранение буферов базисного ФБ;

u_var:= u_input_var & u_int_out_var & u_buf – сохранение переменных, относящихся к базисному ФБ;

u_EI_var – сохранение входных событийных переменных ФБ;

u_EO_var – сохранение выходных событийных переменных ФБ;

u_event_var:= u_EI_var & u_EO_var – сохранение событийных переменных ФБ;

u_disp_var – сохранение переменных диспетчера;

u_count – сохранение счетчиков базисного ФБ;

u_step_environment – сохранение объектов базисного ФБ, не относящихся к выполнению шага алгоритма;

u_EC_action_completion – сохранение объектов базисного ФБ, не относящихся к завершению *EC*-акции.

Приведенный выше список условий сохранения является далеко не полным. Чтобы не путать слово «переменная» в разных контекстах, переменные состояния *SMV*, используемые при моделировании ФБ, в приведенном списке названы объектами. Следует отметить, что можно производить композицию условий для получения новых условий. По сути дела, условия сохранения являются частью перехода и к ним применимы все правила построения переходов.

В разделе спецификаций свойств с использованием логик *LTL* и *CTL* могут быть определены верифицируемые свойства системы. Данный раздел ничем не отличается от аналогичного раздела в предыдущей *SMV*-модели.

4.3. Пример. Модель системы двух АЛУ

В качестве примера системы ФБ рассмотрим систему двух АЛУ (рис. 4.2) [122]. Данный пример предназначен для тестирования предлагаемого метода верификации систем ФБ и содержит все основные элементы базисного ФБ и сети ФБ – входные и выходные события, входные и выходные переменные, внутренние переменные, событийные и информационные связи. Данный пример не претендует на оптимальность, полноту и возможность его использования на практике и носит чисто демонстрационный характер.

Моделируемая система состоит из двух базисных ФБ, представляющих арифметико-логические устройства (АЛУ), и является замкнутой. Выходы одного базисного ФБ соединены с входами другого базисного ФБ. После подачи стартового импульса на вход *initt* блока *alu*1 система должна выполнять бесконечную последовательность действий, состоящую из чередующихся операций сложения и вычитания. Причем выбраны такие входные параметры, что значения переменных не изменяются. Например, если первый блок прибавляет некоторое число, то второй – его же и вычитает. В результате этого пространство состояний системы является ограниченным. На рис. 4.2 кружками обозначены буфера данных *res*1*Buf* и *res*2*Buf*. Они не входят в синтаксическую часть описания системы, но играют важную роль при ее выполнении.

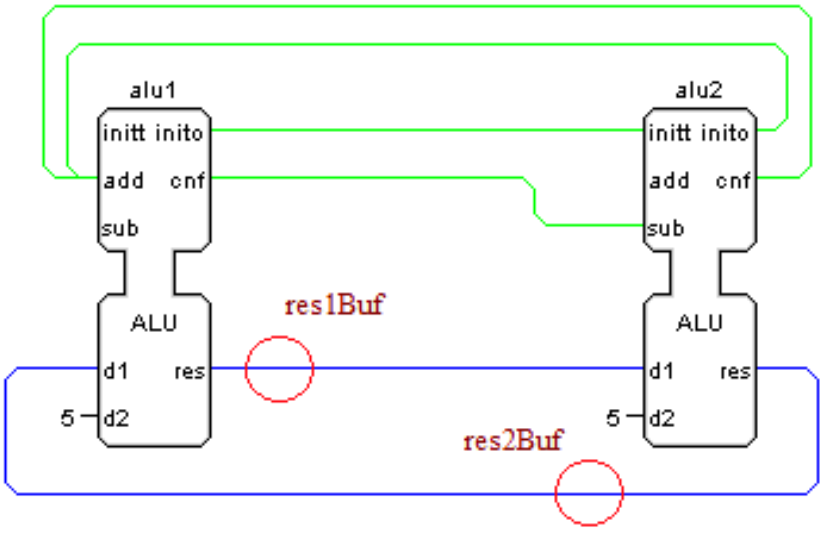


Рис. 4.2. Система двух АЛУ

Базисный ФБ *alu* (рис. 4.3) предназначен для выполнения арифметических действий – сложения и вычитания в зависимости от поступающего на его вход сигнала.

Реализуемые в блоке *alu* алгоритмы:

алгоритм ALG1: n := 13; res := n-10;

алгоритм ALG2: res := d1+d2+n;

алгоритм ALG3: res := d1-d2-n;

где n – внутренняя переменная, инициализируемая числом 13.

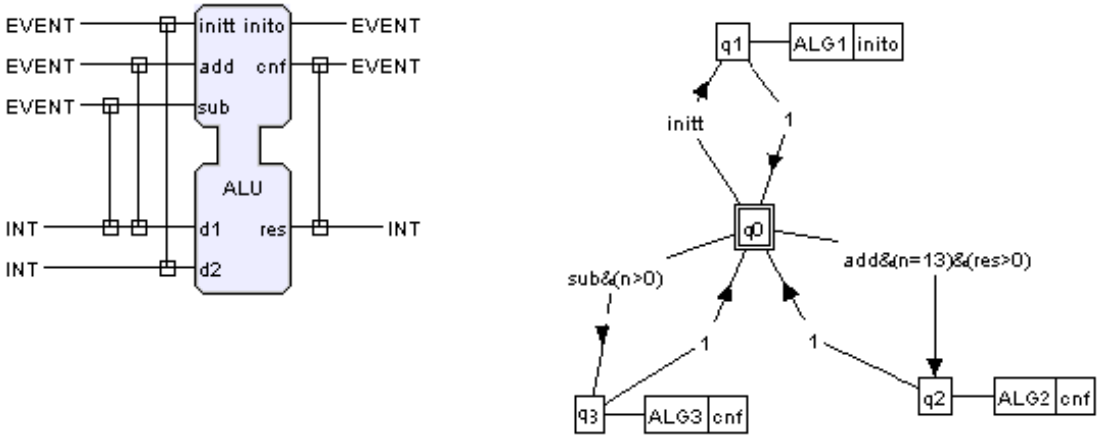


Рис. 4.3. Базисный ФБ ALU:
интерфейс (слева), диаграмма *ECC* (справа)

4.4. Демонстрация подхода 1

Рассмотрим более подробно реализацию на языке *SMV* системы ФБ, представленной на рис. 4.2–4.3 и функционирующей согласно циклической модели выполнения, при использовании метамодели ФБ на основе ФМОСФБ.

Формальная (развернутая) модель, построенная для системы двух АЛУ, включает следующие модули: МСФБ, представляющий систему ФБ верхнего уровня и включающий в свою очередь модуль диспетчера, и два МБФБ *alu*1 и *alu*2 типа *ALU*. Следует обратить внимание на семантический разрыв между метамоделью на основе формализма ФМОСФБ и языком *SMV* в плане типизации модулей: в первом случае она отсутствует, а во втором – присутствует. Поэтому в дальнейшем и предполагается, что формальная модель является развернутой.

Модулям *alu*1 и *alu*2 формальной модели соответствует модуль *SMV* (скорее, тип модуля) под именем *alu*, заголовок которого приведен ниже:

MODULE alu(initt,add,sub,d1_,d2_,inito,cnf,res_, omega, alpha, beta),

где initt, add и sub – входные событийные переменные; d1_ и d2_ –

внешние буфера, связанные с входными переменными d1 и d2 соответственно; inito и cnf – выходные событийные переменные; res_ – внешний буфер, связанный с выходной переменной *res*; omega – переменная окончания передач сигналов в объемлющем ФБ (признак ω); alpha – переменная запуска модуля диспетчером (признак α); beta – переменная окончания работы модуля (признак β). Следует заметить, что формальные параметры модуля внутри модуля системы *Cadence SMV* не объявляются, они определены в объемлющем модуле *SMV*.

Внутренние переменные модуля *SMV*:

```
VAR d1: 0..99; -- Входная переменная d1
    d2: 0..99; -- Входная переменная d2
    res: 0..99; -- Выходная переменная res
    n: 0..99; -- Внутренняя переменная n
    S: {s0,s1,s2}; -- Переменная текущего состояния OSM-машины
    Q: {q0,q1,q2,q3};  -- Переменная текущего состояния ECC-машины
    NA: 0..2;     -- Счетчик EC-акций
    NI: 0..5;  -- Счетчик шагов алгоритма
```

В совокупности параметры модуля *SMV* и внутренние переменные модуля *SMV* представляют все переменные МБФБ *alu*.

Начальные значения переменным задаются в разделе *ASSIGN* модуля *SMV*:

```
ASSIGN
init(Q):= q0; -- Начальное EC-состояние
init(S):= s0; -- Начальное состояние OSM-машины
init(d1):=0;  -- Начальное значение входной переменной d1
init(d2):=0; -- Начальное значение входной переменной d2
init(res):=0; -- Начальное значение выходной переменной
init(n):=0; -- Начальное значение внутренней переменной n
init(NA):=0;  -- Начальное значение счетчика EC-акций
init(NI):=0;  -- Начальное значение счетчика шагов алгоритма
```

Определение условий приоритетного выбора входных сигналов на *SMV*:

```
DEFINE select_initt:= initt;  -- Условие выбора входного сигнала initt
DEFINE select_add:= add & !initt; -- Условие выбора входного
```

сигнала add

DEFINE select_sub:= sub & !initt & !add; -- Условие выбора входного сигнала sub

Определение сторожевых условий *EC*-переходов на *SMV*:

DEFINE GuardCond_q0q1:= 1;

DEFINE GuardCond_q0q2:= (n=13) & (res>0);

DEFINE GuardCond_q0q3:= (n>0);

DEFINE GuardCond_q1q0:= 1;

DEFINE GuardCond_q2q0:= 1;

DEFINE GuardCond_q3q0:= 1;

Определение условий разрешенности *EC*-переходов на *SMV*:

DEFINE EnabledECTran_q0q1:= select_initt & GuardCond_q0q1;

DEFINE EnabledECTran_q0q2:= select_add & GuardCond_q0q2;

DEFINE EnabledECTran_q0q3:= select_sub & GuardCond_q0q3;

DEFINE EnabledECTran_q1q0:= GuardCond_q1q0;

DEFINE EnabledECTran_q2q0:= GuardCond_q2q0;

DEFINE EnabledECTran_q3q0:= GuardCond_q3q0;

Условие существования разрешенных *EC*-переходов на *SMV*:

DEFINE ExistsEnabledECTran:= (Q=q0 & (EnabledECTran_q0q1 | EnabledECTran_q0q2 | EnabledECTran_q0q3)) |(Q=q1 & EnabledECTran_q1q0) | (Q=q2 & EnabledECTran_q2q0) | (Q=q3 & EnabledECTran_q3q0);

Условие отсутствия разрешенных *EC*-переходов на *SMV*:

DEFINE AbsentsEnabledECTran:= (Q=q0 & (!EnabledECTran_q0q1 & !EnabledECTran_q0q2 & !EnabledECTran_q0q3)) | (Q=q1 & !EnabledECTran_q1q0) | (Q=q2 & !EnabledECTran_q2q0) | (Q=q3 & !EnabledECTran_q3q0);

Условия выдачи выходных сигналов на *SMV*:

DEFINE putout_inito:= Q=q2 & NA=1; -- Условие выдачи сигнала inito

DEFINE putout_cnf:= Q=q2 & NA=1 | Q=q3 & NA=1; -- Условие выдачи сигнала cnf

В соответствии со вторым условием выходной сигнал *cnf* выдается, когда завершено выполнение первой *EC*-акции *EC*-состояния *q*2 или первой *EC*-акции *EC*-состояния *q*3.

Примем, что в данной модели системы ФБ будет использоваться дисциплина *DelEI*1 для сброса входных сигналов, регламентируемая правилами типа $p_{EI}^{B,1}$ и $p_{EI}^{B,2}$ формальной модели, представ-

ленной в разделе 3. Закодируем на языке *SMV* данные правила:

```
next(initt):= case  -- Сброс событийного входа initt
(S=s1 & select_initt) | (alpha & S=s0 & !select_initt): 0;
1: initt;
esac;
next(add):= case  -- Сброс событийного входа add
(S=s1 & select_add) | (alpha & S=s0 & !select_add): 0;
1: add;
esac;
next(sub):= case  -- Сброс событийного входа sub
(S=s1 & select_sub) | (alpha & S=s0 & !select_sub): 0;
1: sub;
esac;
```

Съем входных данных определяется правилом типа $p_{VI}^{B,1}$ формальной модели и представляется на *SMV* следующим образом:

```
next(d1):= case  -- Съем данных с входа d1
alpha & S=s0 & (select_add | select_sub): d1_;
1: d1;
esac;
next(d2):= case  -- Съем данных с входа d2
alpha & S=s0 & select_initt: d2_;
1: d2;
esac;
```

Приоритетная группа правил $p_Q^{B,1}$ изменения *EC*-состояния кодируется на языке *SMV* в форме *case*-оператора:

```
next(Q):= case
  Q=q0 & S=s1 & EnabledECTran_q0q1: q1; -- EC-переход с наивысшим приоритетом
  Q=q0 & S=s1 & EnabledECTran_q0q2: q2;
  Q=q0 & S=s1 & EnabledECTran_q0q3: q3;
  Q=q1 & S=s1 & EnabledECTran_q1q0: q0;
  Q=q2 & S=s1 & EnabledECTran_q2q0: q0;
  Q=q3 & S=s1 & EnabledECTran_q3q0: q0;  -- EC-переход с наинизшим приоритетом
  1 : Q;
esac;
```

Приоритет *EC*-переходов из одного и того же *EC*-состояния (например, $q0$) определяется положением соответствующей пары «Условие-Действие»: чем выше пара стоит в описании *case*-оператора, тем выше приоритет.

Правила переходов состояний *OSM*-машины $p_S^{B,1}$, $p_S^{B,2}$, $p_S^{B,3}$

è $p_S^{B,4}$ формальной модели кодируется так:

```
next(S):= case
  alpha & S=s0 & (select_initt | select_add | select_sub): s1;
  S=s1 & ExistsEnabledECTran: s2;
  S=s2 & NA=0 : s1;
  S=s1 & AbsentsEnabledECTran: s0;
  1 : S;
esac;
```

Данное правило довольно прозрачное и полностью соответствует Стандарту. Условие $NA = 0$ определяет, что выполнение всех *EC*-акций в *EC*-состоянии закончено.

Правила изменения счетчика *EC*-акций $p_{NA}^{B,1}, p_{NA}^{B,2}$ è $p_{NA}^{B,3}$ на языке *SMV*:

```
next(NA):= case
   S=s1: 1;
   S=s2 & NI=0 & ((Q=q0 & NA<1) | (Q=q1 & NA<1) | (Q=q2 & NA<1) | (Q=q3 & NA<1)): NA+1;
   S=s2 & NI=0 & ((Q=q0 & NA=1) | (Q=q1 & NA=1) | (Q=q2 & NA=1) | (Q=q3 & NA=1)): 0;
   1: NA;
esac;
```

Данными правилами для каждого *EC*-состояния определено, когда счетчик *EC*-акций увеличивается на единицу, а когда сбрасывается. Видно, что счетчик *EC*-акций увеличивается на единицу, когда в *OSM*-состоянии *s*2 закончено выполнение всех шагов текущего алгоритма ($NI = 0$).

Правила изменения счетчика шагов алгоритма $NI < p_{NI}^{B,4}, p_{NI}^{B,5}, p_{NI}^{B,6}, p_{NI}^{B,7} >$ на языке *SMV*:

```
next(NI):= case
   S=s1: 1;
   S=s2 & (Q=q0 & NA=1 & NI<1 | Q=q1 & NA=1 & NI<2 | Q=q2 & NA=1 & NI<1 | Q=q3 & NA=1 & NI<1): NI+1;
   S=s2 & (Q=q0 & NA=1 & NI=1 | Q=q1 & NA=1 & NI=2 | Q=q2 & NA=1 & NI=1 | Q=q3 & NA=1 & NI=1): 0;
   0: 1;
   1: NI;
esac;
```

Счетчик *NI* увеличивается на единицу, если в текущей *EC*-ак-

ции текущего *EC*-состояния выполнен не последний шаг алгоритма, и сбрасывается в ноль – если выполнен последний шаг. Единичное значение счетчик *NI* принимает в *OSM*-состоянии *s*1.

Выполнение алгоритмов (правила $p_{VO}^{B,1}$ è $p_{VV}^{B,1}$) рассмотрим на примере изменения выходной переменной *res* на языке *SMV*:

```
next(res):= case
  S=s2 & Q=q1 & NA=1 & NI=2: (n - 10) mod 90;
  S=s2 & Q=q2 & NA=1 & NI=1: (d1 + d2  + n) mod 90;
  S=s2 & Q=q3 & NA=1 & NI=1: (d1 - d2  - n) mod 90;
  1 : res;
esac;
```

Как видно, изменение переменной *res* может производиться в разных *EC*-состояниях, разных *EC*-акциях и разных шагах. Деление результата вычислений по модулю (в данном случае 90) делается, чтобы предотвратить неконтролируемый рост значения переменной при первоначальном исследовании системы, когда поведение системы неизвестно.

Пример кодировки правила (типа $p_{EO}^{B,1}$) выдачи выходного сигнала *cnf*:

```
next(cnf):= case
   S=s2 & NI=0 & putout_cnf: 1;
   1 : cnf;
esac;
```

Следует заметить, что в данном случае выходной сигнал выдается немедленно – в состоянии *s*2 *OSM*-машины.

Выдача выходных данных (через выход *res*) представлена одним правилом типа $p_{VOB}^{B,1}$ и моделируется следующим фрагментом *SMV*-кода:

```
next(res_):= case
   S=s2 & NI=0 & & putout_cnf: res;
   1 : res_;
esac;
```

Как можно заметить, антецедент данного правила равен антецеденту вышеприведенного правила, что означает, что данные действия выполняются при выдаче сигнала *cnf*. В соответствии с данным правилом в выходной буфер *res_* переписывается значение выходной переменной *res*.

Диспетчирование выполнения ФБ регулируется управляющими

переменными α и β. Установка переменной окончания работы ФБ β определяется правилами $p_{\beta}^{B,1}$ è $p_{\beta}^{B,2}$ формальной модели и представляется *case*-оператором:

```
next(beta):= case
(alpha & omega & S=s0 & (!select_initt & !select_add & !select_sub)
| S=s1 & AbsentsEnabledECTran): 1;
1: beta;
esac;
```

Сброс переменной запуска МБФБ определяется правилом $p_{\alpha}^{B,1}$ и кодируется как:

```
next(alpha):= case
(alpha & omega & S=s0 & (!select_initt & !select_add & !select_sub)
| S=s1 & AbsentsEnabledECTran): 0;
1: alpha;
esac;
```

Как можно видеть, переменные *alpha* и *beta* работают в противофазе. Изменение переменных производится в двух случаях: 1) в случае «холостого» запуска ФБ, когда на его входах нет сигналов; 2) в случае нормального завершения выполнения ФБ, когда не осталось разрешенных *EC*-переходов.

Модуль составного ФБ, входящий в состав формальной модели и соответствующий описанию, приведенному на рис. 4.2, реализуется как главный модуль *SMV*, не имеющий входов и выходов:

```
MODULE main
```

Формальная модель составного ФБ включает два компонентных базисных ФБ (типа *alu*): *alu*1 и *alu*2. На языке *SMV* они описываются следующим образом:

```
VAR alu1: process alu(initt1, add1, sub1, res2Buf,  vc1, inito1, cnf1, res1Buf, omega, alpha1, beta1);
VAR alu2: process alu(initt2, add2, sub2, res1Buf,  vc2, inito2, cnf2, res2Buf, omega, alpha2, beta2);
```

Кроме того, в МСФБ входит модуль диспетчера, описываемый как асинхронный процесс в *SMV*:

```
VAR disp: process schedulerCyclic(alpha1, alpha2, beta1, beta2);
```

Из формальной метамодели МСФБ (подраздел 3.8.1) следует, что у каждого компонентного ФБ имеются интерфейсные переменные, представляющие событийные и информационные входы-

выходы ФБ, а также дополнительные переменные времени выполнения. Данные переменные представлены переменными состояния модуля *SMV* (рис. 4.4). По возможности этим переменным давались имена, созвучные именам из предметной области – системы ФБ на рис. 4.2. Следует также заметить, что переменным компонентного ФБ соответствуют определенные переменные соответствующего ему модуля (реального экземпляра), представленные как параметры (см. рис. 3.12). Например, переменным initt1, add1 и sub1 (из рис. 4.4) соответствуют входные событийные переменные initt, add и sub модуля *alu*1.

```
VAR initt1: boolean;          VAR initt2: boolean;
    add1: boolean;                add2: boolean;
    sub1: boolean;                sub2: boolean;
    inito1:boolean;               inito2:boolean;
    cnf1: boolean;                cnf2: boolean;
    vc1: 0..5;                    vc2: 0..5;
    alpha1:boolean;               alpha2:boolean;
    beta1: boolean;               beta2: boolean;
    res1Buf: 0..99;               res2Buf: 0..99;
```

Рис. 4.4. Переменные *SMV*-модуля для компонентных ФБ: для *alu*1 (слева) и *alu*2 (справа)

Все переменные, относящиеся к компонентным ФБ, инициализируются нулями, за исключением следующих переменных:

```
init(alpha1):= 1;
init(vc1):=5;
init(initt1):= 1;
```

Переменная *alpha*1 задает стартовый сигнал от диспетчера, переменная *initt* – (начальное) входное событие на входе *initt* ФБ, переменная *vc*1 представляет константу 5. В принципе можно обойтись и без этой переменной, но она используется для унификации описания.

Основной функционал системы выполняют правила для передвижения сигналов от выходов одних ФБ к входам других. Ниже приведена кодировка на *SMV* правила типа $p_{EI^j}^{C,C,1}$ для передачи сигнала от выхода *cnf* блока *alu*1 к входу *sub* блока *alu*2.

```
next(sub2):= case
  cnf1  : 1;
```

```
1 :  sub2; esac;
```

Данное правило устанавливает значение переменной *sub*2 в единицу.

Для сброса источников сигналов в МСФБ используются правила типа $p_{EO^j}^{C,C,1}$. Например, для сброса выходной событийной переменной *cnf* блока *alu*1 данное правило кодируется следующим образом:

```
next(cnf1):=0;
```

Оно сбрасывает переменную *cnf*1 в ноль. Следует отметить, что для данного случая правила $p_{EI^j}^{C,C,1}$ и $p_{EO^j}^{C,C,1}$ были модифицированы путем удаления из их антецедентов переменной α, поскольку моделируется система самого верхнего уровня.

В МСФБ, кроме того, определено условие завершения передач сигналов, используемое в компонентных базисных ФБ. Передачи сигналов считаются завершенными, когда все источники сигналов пусты. На *SMV* это условие кодируется следующим образом:

```
DEFINE omega:= !inito1 & !cnf1 & !inito2 & !cnf2;
```

Модуль (главного) диспетчера является небольшим и его текст на языке *SMV* приведен ниже полностью (в две колонки):

```
MODULE
schedulerCyclic (alpha1,
alpha2, beta1, beta2)
ASSIGN
next(alpha1):= case
beta2: 1;
1: alpha1;
esac;
next(alpha2):= case
beta1: 1;
1: alpha2;
esac;
next(beta1):= case
beta1: 0;
1: beta1;
esac;
next(beta2):= case
beta2: 0;
1: beta2;
esac;
```

Логика работы диспетчера проста – по окончании работы первого ФБ запускается второй, и наоборот.

Для анализа модели может использоваться временная логика *CTL* [63], реализованная в *SMV*. При этом с использованием временных операторов *AX*, *AF*, *AG*, *EX*, *EF*, *EG* и *U* могут быть заданы проверяемые свойства системы ФБ.

Примеры запросов на достижимость:

*SPEC EF beta*2 – «Возможно ли, что когда-либо ФБ *alu*2 закон-

чит свое выполнение ?» (*true*);

SPEC EF (*res*1*Buf*=18) – «Может ли значение выходного буфера блока *alu*1 принимать значение 18?» (*true*).

Пример запросов на живость (шаблон *Possibility* [94]):

SPEC AG(*EF* (*alu*1.*Q*=*q*2)) – «Повторяется ли *EC*-состояние *q*2 блока *alu*1 при функционировании бесконечно часто, иными словами, является ли состояние *q*2 блока *alu*1 живым ?» (*true*)

Пример запроса другого типа (шаблон *Eventual Response* [94]):

SPEC AG (*alpha*2 -> *AF alpha*1) – «Всегда ли запуск первого блока неизбежно влечет за собой запуск второго блока ?» (*true*).

Следует отметить, что для построения *CTL*-запросов могут быть использованы шаблоны. Шаблоны для общих свойств (живость, безопасность, повторяемость, наличие циклов и т.д.) хорошо известны из литературы [63].

4.5. Демонстрация подхода 2

Рассмотрим более подробно реализацию на языке *SMV* системы ФБ, представленной на рис. 4.2, 4.3 и функционирующей согласно циклической модели выполнения, при использовании метамодели ФБ на основе структуры Крипке. Как и в предыдущем случае, при иллюстрации подхода полная формальная модель системы ФБ ввиду ее громоздкости не строится, вместо этого демонстрируется связь элементов метамодели с элементами *SMV*-модели.

Поскольку система двух АЛУ довольно проста, то несложно «раскрыть» эту систему и построить соответствующую систему экземпляров вручную. Будем давать элементам глобальные имена в соответствии со способом, отмеченным выше. Поскольку моделируемая система является двухуровневой, то суффиксом имен элементов будет имя экземпляра ФБ (*alu*1 или *alu*2). Переменные, относящиеся к МБФБ *alu*1, можно объявить на *SMV* следующим образом:

```
VAR d1_alu1: 0..99;
    d2_alu1: 0..99;
    res_alu1: 0..99;
    n_alu1: 0..99;
    S_alu1: {s0,s1,s2};
    Q_alu1: {q0,q1,q2,q3};
    NA_alu1: 0..2;
    NI_alu1: 0..5;
```

```
VAR initt_alu1: boolean;
    add_alu1: boolean;
    sub_alu1: boolean;
    inito_alu1:boolean;
    cnf_alu1: boolean;
    resBuf_alu1: 0..99;
    alpha_alu1:boolean;
    beta_alu1: boolean;
```

Причем в левом столбце объявлены внутренние переменные МБФБ, а в правом – внешние. Переменные, относящиеся к экземпляру МБФБ *alu*2, объявляются аналогично.

Раздел начальных значений переменных выглядит следующим образом:

INIT

(Q_alu1=q0 & S_alu1=s0 & d1_alu1=0 & d2_alu1=0 & res_alu1=0 & n_alu1=0 & NA_alu1=0 & NI_alu1=0 & Q_alu2=q0 & S_alu2=s0 & d1_alu2=0 & d2_alu2=0 & res_alu2=0 & n_alu2=0 & NA_alu2=0 & NI_alu2=0 & alpha_alu1=1 & beta_alu1=0 & initt_alu1=1 & add_alu1=0 & sub_alu1=0 & inito_alu1=0 & cnf_alu1=0 & resBuf_alu1=0 & alpha_alu1=1 & beta_alu1=0 & alpha_alu2=0 & beta_alu2=0 & initt_alu2=0 & add_alu2=0 & sub_alu2=0 & inito_alu2=0 & cnf_alu2=0 & resBuf_alu2=0 & alpha_alu2=0 & beta_alu2=0)

В качестве примера ниже приводятся определения переходов разных типов на языке *SMV*. Приведенные описания являются вполне самоопределяемыми.

Переход «Срабатывание *EC*-перехода $q0 \to q1$ в экземпляре ФБ *alu*1» типа $T_{FT}^{B,1}$ формальной модели представляется следующим фрагментом *SMV*:

S_alu1=s1 & Q_alu1=q0 & EnabledECTran_q0q1_alu1 & -- Условие разрешенности перехода

next(Q_alu1)=q1 & next(S_alu1)=s2 & next(initt_alu1)=0 & next(add_alu1)=0 & next(sub_alu1)=0 & next(NA_alu1)=1 & next(NI_alu1)=1 & -- Действия перехода

u_var_alu1 & u_EO_var_alu1 & u_disp_var_alu1 & u_obj_alu2 -- Сохранение значений переменных

Переход «Синхронный съем данных в экземпляре ФБ *alu*1» типа $T_{SD}^{B,1}$ представляется двумя *SMV*-переходами:

alpha_alu1 & S_alu1=s0 & ((add_alu1 | sub_alu1) & !initt) & -- Условие разрешенности перехода

next(d1_alu1)=resBuf_alu2 & next(S_alu1)=s1 & -- Действия перехода

next(d2_alu1)=d2_alu1 & u_count_alu1 & u_event_var_alu1 & u_int_out_var_alu1 & u_buf_alu1 & u_disp_var_alu1 & u_obj_alu2 -- Сохранение значений переменных

|

alpha_alu1 & S_alu1=s0 & initt_alu1 & -- Условие разрешенно-

сти перехода

next(d2_alu1)=5 & next(S_alu1)=s1 & -- Действия перехода

next(d1_alu1)=d1_alu1 & u_count_alu1 & u_event_var_alu1 & u_int_out_var_alu1 & u_buf_alu1 & u_disp_var_alu1 & u_obj_alu2 -- Сохранение значений переменных

Следует отметить, что переходы $T_{SD}^{B,1}$ формальной модели в приведенном выше варианте для простоты реализованы частично: в данном случае не рассматриваются нереальные комбинации входных сигналов из $\mathrm{Pr}_2\, f_{EV}$, например, когда одновременно приходят сигналы *initt* и *add* или *sub*. Первый приведенный *SMV*-переход ориентирован только на съем данных с информационного входа *d*1, а второй – на съем данных с *d*2.

Переход «Выполнение (безусловного) шага 1 алгоритма в первой *EC*-акции *EC*-состояния *q*2 в экземпляре ФБ *alu*2» типа $T_{EX}^{B,1}$:

Q_alu2=q2 & NA_alu2=1 & NI_alu2=1 & -- Условие разрешенности перехода

next (res_alu2)=(d1_alu2 + d2_alu2 + n_alu2) & next(NI_alu2)=0 & -- Действия перехода

next(n_alu2)=n_alu2 & u_step_environment_alu2 -- Сохранение значений переменных

Переход «Завершение *EC*-акций (*q*2,1) и (*q*3,1) с выдачей выходного сигнала *cnf* и выдачей данных с выхода *res* в экземпляре ФБ *alu2*» типа $T_{CA}^{B,2}$:

S_alu2=s2 & NI_alu2=0 & (Q_alu2=q2 & NA_alu2=1 | Q_alu2=q3 & NA_alu2=1) & -- Условие разрешенности перехода

next(cnf_alu2)=1 & next(resBuf_alu2)=res_alu2 & next(NA_alu2)=0 & -- Действия перехода

next(inito_alu2)=inito_alu2 & u_EC_action_completion_alu2 -- Сохранение значений переменных

Переход «Завершение выполнения *EC*-состояния в экземпляре ФБ *alu*1» типа $T_{CS}^{B,1}$:

S_alu1=s2 & (Q_alu1=q1 & NA_alu1>1 | Q_alu1=q2 & NA_alu1>1 | Q_alu1=q3 & NA_alu1>1) & -- Условие разрешенности перехода

next(S_alu1)=s1 & next(NA_alu1)=1 & next(NI_alu1)=1 & -- Действия перехода

next(Q_alu1)=Q_alu1 & u_var_alu1 & u_event_var_alu1 & u_disp_var_alu1 & u_obj_alu2 -- Сохранение значений переменных

Переход «Завершение выполнения базисного ФБ *alu*1» ти-

па $T_{CF}^{B,1}$:

S_alu1=s1 & AbsentsEnabledECTran_alu1 & -- Условие разрешенности перехода

next(S_alu1)=s0 & next(alpha_alu1)=0 & next(beta_alu1)=1 & -- Действия перехода

u_var_alu1 & u_EI_var_alu1 & u_EO_var_alu1 & u_count_alu1 & u_obj_alu2 -- Сохранение значений переменных

Переход «Пустой запуск базисного ФБ *alu*1» типа $T_{ES}^{B,1}$:

alpha_alu1 & S_alu1=s0 & !select_initt_alu1 & !select_add_alu1 & !select_sub_alu1 & -- Условие разрешенности перехода

next(alpha_alu1)=0 & next(beta_alu1)=1 & -- Действия перехода

next(S_alu1)=S_alu1 & u_var_alu1 & u_event_var_alu1 & u_count_alu1 & u_obj_alu2 -- Сохранение значений переменных

Переход «Передача сигнала с выхода *inito* или выхода *cnf* экземпляра ФБ *alu*2 на вход *add* экземпляра *alu*1» типа $T_{TO}^{C,1}$:

(inito_alu2 | cnf_alu2) & -- Условие разрешенности перехода

next(add_alu1)=1 & next (inito_alu2)=0 & next(cnf_alu2)=0 & -- Действия перехода

next(initt_alu1)=initt_alu1 & next(sub_alu1)=sub_alu1 & & u_signal_transfer_from_alu2 -- Сохранение значений переменных

Переход «Запуск компонентного ФБ *alu*2» типа $T_{SF}^{D,2}$:

beta_alu1 & omega & -- Условие разрешенности перехода

next(alpha_alu2)=1 & next(beta_alu1)=0 & -- Действия перехода

next(alpha_alu1)=alpha_alu1 & next(beta_alu2)=beta_alu2 & u_all_exept_disp -- Сохранение значений переменных

Ниже приводятся примеры условий сохранения значений для определенных групп переменных, используемых в приведенных выше правилах.

Условие сохранения выходных событийных переменных экземпляра ФБ *alu*1:

DEFINE u_EO_var_alu1:= next(inito_alu1)= inito_alu1 & next(cnf_alu1)=cnf_alu1;

Условие сохранения событийных переменных ФБ экземпляра ФБ *alu*1:

DEFINE u_event_var_alu1:= u_EI_var_alu1 & u_EO_var_alu1;

Условие сохранения переменных диспетчера:

DEFINE u_disp_var_alu1:= next(alpha_alu1)=alpha_alu1 & next(beta_alu1)=beta_alu1;

Условие сохранения объектов базисного ФБ, не относящихся к выполнению шага алгоритма ФБ *alu*1:

DEFINE u_step_environment_alu1:= next(S_alu1)=S_alu1 & next(Q_alu1)=Q_alu1 & next(NA_alu1)=NA_alu1 & u_input_var_alu1 & u_ext_obj_alu1 & u_obj_alu2;

Условие сохранения всех объектов экземпляра базисного ФБ *alu*2:

DEFINE u_obj_alu2:= u_inner_obj_alu2 & u_ext_obj_alu2;

Ниже приводятся примеры условий, определяющих логику функционирования ФБ.

Условие приоритетного выбора входного сигнала *sub* экземпляра *alu*1:

DEFINE select_sub_alu1:= sub_alu1 & !initt_alu1 & !add_alu1;

Сторожевое условие *EC*-перехода $q0 \rightarrow q2$ экземпляра *alu*1:

DEFINE GuardCond_q0q2_alu1:= (n_alu1=13) & (res_alu1>0);

Условие разрешенности *EC*-перехода $q0 \rightarrow q3$ экземпляра *alu*2:

DEFINE EnabledECTran_q0q3_alu2:= select_sub_alu2 & GuardCond_q0q3_alu2;

Условие существования разрешенных *EC*-переходов в экземпляре *alu*1:

DEFINE ExistsEnabledECTran_alu1:= (Q_alu1=q0 & (EnabledECTran_q0q1_alu1 | EnabledECTran_q0q2_alu1 | EnabledECTran_q0q3_alu1)) |(Q=q1 & EnabledECTran_q1q0_alu1) | (Q=q2 & EnabledECTran_q2q0_alu1) | (Q=q3 & EnabledECTran_q3q0_alu1);

Условие отсутствия разрешенных *EC*-переходов в экземпляре *alu*2:

DEFINE AbsentsEnabledECTran_alu2:= (Q_alu2=q0 & (!EnabledECTran_q0q1_alu2 & !EnabledECTran_q0q2_alu2 & !EnabledECTran_q0q3_alu2)) | (Q_alu2=q1 & !EnabledECTran_q1q0_alu2) | (Q_alu2=q2 & !EnabledECTran_q2q0_alu2) | (Q_alu2=q3 & !EnabledECTran_q3q0_alu2);

Условие завершения передач сигналов в системе:

DEFINE omega:= !inito_alu1 & !cnf_alu1 & !inito_alu2 & !cnf_alu2.

5. Графотрансформационный подход к синтезу формальных моделей систем функциональных блоков

Одним из слабо формализованных этапов разработки управляющих систем на основе ФБ стандарта IEC 61499 является разработка формальных моделей контроллеров и объектов управления. Это приводит к ошибкам проектирования, усложняет автоматизацию данного этапа и замедляет процессы разработки и повторного проектирования системы в целом. В данном разделе для синтеза формальных моделей систем ФБ IEC 61499 предлагается подход на основе *управления моделями*. Основной целью данного раздела является разработка правил трансформации моделей ФБ в формальные модели, которые в дальнейшем можно исследовать. В качестве целевой модельной формы были выбраны арифметические *NCES*-сети (*aNCES*-сети), хорошо подходящие для представления систем ФБ [28]. Как было показано в [16], для их анализа можно использовать метод *Model checking*. Прототип системы синтеза сетевых моделей ФБ строится на основе общецелевой системы трансформации графов *AGG* [222]. Следует заметить, что ранее предпринимались попытки построения правил преобразования систем ФБ в формализм *NCES*-се-тей, однако при этом правила определялись неформально и несистематизированно [166].

5.1. Краткие сведения из области трансформации графов

Существует несколько подходов к перезаписи графов, среди которых – алгебраический подход. Алгебраический подход в свою очередь подразделяется на три разновидности: *SPO* (*single-pushout*), *DPO* (*double-pushout*) и *pullback* [123, 149]. Мы кратко рассмотрим первый метод, поскольку он реализован в системе *AGG*, используемой нами в дальнейшем для трансформации.

Пусть L, R, G и H будут помеченные графы. Прямая трансформация графа $G \Rightarrow {}_{t}H$ определяется парой $t = (p, m)$, состоящей из правила замены графов $p{:}L \to R$ и инъективного графового морфизма (называемого соответствием) $m{:}\ L \to G$. Используя коммутативную диаграмму (*pushout*) (рис. 5.1) можно вычислить морфизмы $m'{:}R \to H$ и $p'{:}G \to H$. Последовательность $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ прямых преобразований графа называется преобразованием графа и обозначается $G_0 \Rightarrow^{*} G_n$.

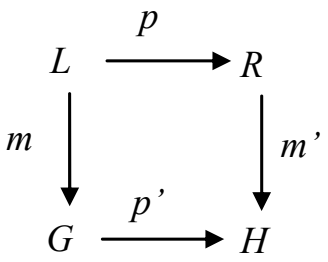


Рис. 5.1. Схематичное представление прямого вывода
в виде универсального квадрата

Отрицательное условие применения (*NAC*-условие) для правила p – это графовый морфизм $nac: L \to \bar{L}$. Прямая трансформация графа $G \Rightarrow_{(p,m)} H$ удовлетворяет *NAC*-условию, если не существует графовый морфизм $\bar{m}: \bar{L} \to G$ такой, что $\bar{m} \bullet nac = m$. Упрощенно говоря, *NAC*-условие – это граф, который определяет запрещенную графовую структуру. К одному правилу может быть присоединено несколько *NAC*-условий. В этом случае правило применимо, если удовлетворяются все *NAC*-условия.

Атрибутный граф – это граф, вершины и дуги которого помечены абстрактными типами данных. В случае атрибутных графов для применимости правила необходимо также выполнение условий по атрибутам вершин и дуг. При выполнении правила производится передача и вычисление атрибутов определенной части результирующего графа. Конфликт между трансформациями графов может быть найден путем определения критических пар правил замены графов.

Графом типов (*type graph*) называется некоторый фиксированный граф *TG*, представляющий информацию о типах узлов и дуг графов, генерируемых графовой грамматикой (ГГ). Назначение типов графу *G* с помощью графа *TG* задается полным графовым морфизмом $t: G \to TG$. Если существует графовый морфизм *t* для *G*, то граф *G* называется типизированным. Ограничения на структуру взаимосвязи экземпляров типов узлов могут задаваться с помощью кардинальности (множественности) роли связи. Кроме того, можно также задавать и множественность узлов.

Более подробную информацию о трансформации атрибутных графов можно найти в [134].

5.2. Поток моделей в процессе синтеза

Представим процесс синтеза *aNCES*-сетей для систем ФБ как поток графовых моделей (рис. 5.2) [19, 20].

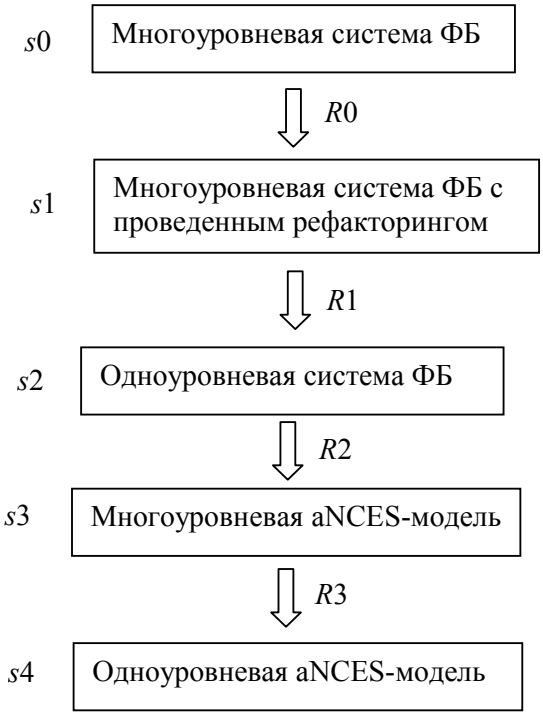


Рис. 5.2. Поток моделей в задаче синтеза NCES-сетей для систем ФБ

Модели в приведенной на рис. 5.2 диаграмме состояний представлены как состояния, а макропреобразования – как переходы. Отправным пунктом преобразований является графовая модель исходной системы ФБ (состояние *s*0). Как правило, эта модель является многоуровневой. Конечным пунктом преобразования является графовая модель одноуровневой («плоской») *aNCES*-сети (состояние *s*4). Эта модель является наиболее приемлемой для дальнейшего анализа. Промежуточное (опциональное) состояние *s*1 представляет исходную систему ФБ, в базовых ФБ которой проведен рефакторинг диаграмм управления выполнением (диаграмм *ECC*) [246], рассмотренный подробно в следующем разделе. Состояние *s*2 определяет одноуровневую («плоскую») систему ФБ, которую можно принять за некоторую нормализованную форму для дальнейшего использования, например, для реализации имитационного моделирования или анализа. Состояние *s*3 представляет многоуровневую *aNCES*-модель, состоящую из множества взаимосвязанных *aNCES*-модулей. Это наиболее компактное и понятное для пользователя представление *aNCES*-моделей. Состояние *s*4, определяющее одноуровневую («плоскую») *aNCES*-сеть, наиболее подходяще для исследования сетевой модели, например, с помощью метода имитационного моделирования или проверки моделей [16]. Переход из од-

ного состояния в другое производится в результате применения наборов трансформационных правил $R0$, $R1$, $R2$ и $R3$.

5.3. Моделирование систем ФБ на основе арифметических *NCES*-сетей

Формализм арифметических *NCES*-сетей

Как было отмечено в [137, 197], описательных возможностей *NCES*-сетей недостаточно для описания сложных ФБ. Для моделирования систем ФБ ниже предлагаются модульные арифметические *NCES*-сети (*aNCES*-сети), являющиеся расширением обычных *NCES*-сетей [232]. В отличие от последних в *aNCES*-сетях могут использоваться целочисленные и нецелочисленные маркировки и веса дуг, а также функции обработки данных. Так же, как и в сетях Петри с двойственными переходами (СПДП) [254], с помощью *aNCES*-сетей легко представляются условные операторы и циклы в спецификациях встроенных систем. В отличие от СПДП маркировка позиции в *aNCES*-сетях элементарна, однако ее интерпретация (число меток или значение данных) зависит от типа дуги, исходящей из соответствующей позиции.

В зависимости от типа маркировки и весов дуг будем различать арифметические *NCES*-сети целочисленного и вещественного типов, причем второй тип сети включает первый. Арифметические *NCES*-сети вещественного типа определяются как

$$N = (P, T, H, A, B, I, W, S, M, F, Q, C, m_0),$$

где P – конечное непустое множество позиций; T – конечное непустое множество переходов; $H \subseteq P\times T \cup T\times P$ – множество обычных дуг; $A \subseteq P\times T \cup T\times P$ – множество арифметических дуг, $H \cap A = \varnothing$; $B \subseteq P\times T$ – множество условных дуг; $I \subseteq P \times T$ – множество ингибиторных дуг, $B \cap I = \varnothing$; $W{:}H \cup B \to R$ – функция весов обычных и условных дуг, где R – множество действительных чисел; $S \subseteq T \times T$ – множество событийных дуг; $M{:}T\to\{\&,\vee\}$ – режим срабатывания перехода; $F = F^{(0)} \cup F^{(1)} \cup F^{(2)} \cup\ldots$ – бесконечное множество функций различных арностей. Функция $f\in F^{(n)}$ определяется как $f{:}R^n\to R$; $Q{:}T\to F \cup \{none\}$ – функция, назначающая переходам функции из F; $C{:}\ A \cap P\times T\to\{1, 2, \ldots\}$ – функция нумерации входных арифметических дуг переходов; $m_0{:}P \to R$ – начальная маркировка сети.

Введем следующие множества: $apre^t = \{p \mid (p,t)\in A\}$ – множество входных арифметических позиций перехода $t\in T$; $apost^t = \{p \mid (t, p) \in \in A\}$ – множество выходных арифметических позиций перехода

$t \in T$. Должны выполняться следующие ограничения: 1) $\forall t \in T$ [$|apost^t| \le 1$]; 2) $|apre^t|$ должно быть равно арности функции $Q(t)$ (если таковая имеется).

Функция нумерации входных арифметических дуг перехода t определяется как $C^t: apre^t \to \{1, 2, \ldots, |apre^t|\}$. Общая функция нумерации C вычисляется как $\tilde{N} = \bigcup_{t \in T} \tilde{N}^t$. Порядок нумерации входных арифметических дуг должен соответствовать порядку аргументов функции, поставленной в соответствие переходу.

Семантика вещественных *aNCES* (без функций) сходна с семантикой обычных *NCES*, несмотря на то, что используется вещественный домен чисел. Семантика вещественных *aNCES* (с функциями) отличается от семантики обычных *NCES* в следующем: 1) маркировка позиций из множеств $apre^t$ и $apost^t$ не влияет на разрешенность перехода $t \in T$; 2) при срабатывании перехода $t \in T$, для которого $Q(t) \neq none$, маркировка позиций $apre^t$ не изменяется, а маркировка позиции из $apost^t$ устанавливается равной значению функции $Q(t)$. При вызове функции $Q(t)$ позиции из $apre^t$ упорядочиваются в соответствии с функцией нумерации C^t.

Возможно создание доменно-ориентированных расширений *aNCES*-сетей в целом и переходов в отдельности для конкретных предметных областей. Например, для моделирования алгоритмов полезны переходы-обработчики, выполняющие элементарные арифметические, логические операции и операции сравнения (рис. 5.3). Из них могут быть составлены любые алгоритмы обработки данных.

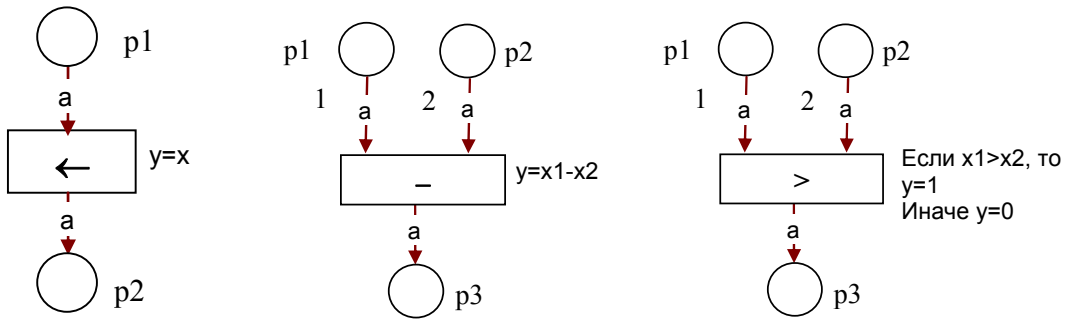


Рис. 5.3. Доменно-ориентированные переходы:
оператор присваивания (слева); оператор вычитания (в центре),
оператор сравнения (справа)

Другое важное расширение *aNCES*-сетей находится в области структуризации сетевых моделей. Известно, что обычные модульные *NCES*-сети не позволяют определить потоки меток между модулями [181]. Это затрудняет моделирование потоков данных меж-

ду ФБ. Используемый в *aNCES*-сетях модуль представляет собой макроопределение (рис. 5.4). В отличие от *NCES*-модулей между модулями этого типа возможна передача меток. Тип входа-выхода модуля определяется типом проходящей через границу модуля дуги. Тип дуги при этом определяется типом дуги по определению и парой «источник-приемник» дуги. На рис. 5.4 используются следующие обозначения: *e* – событийная дуга, *c* – условная дуга, *i* – ингибиторная дуга, *pa* – арифметическая дуга типа «позиция-переход», *tp* – простая дуга типа «переход-позиция» и т.д. В дальнейшем все арифметические дуги будут помечаться символом *a*.

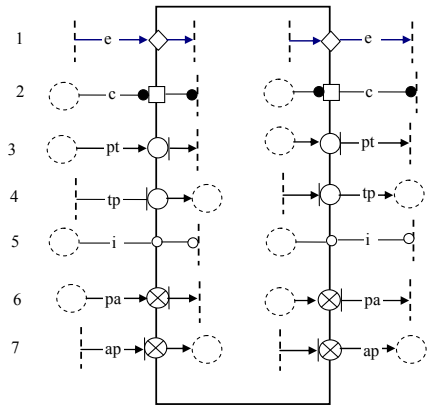


Рис. 5.4. Схема модуля *aNCES*

Моделирование управления выполнением операций в базисном функциональном блоке

Для моделирования управления выполнением операций в базисном ФБ IEC 61499 используется *NCES*-модуль *ControlECC*. Модули этого типа создаются по одному на каждый экземпляр базисного ФБ. По сути, они представляют управляющую часть интерпретатора *ECC* (в терминах стандарта IEC 61499 – *OSM*-машину). Интерфейс *NCES*-модуля *ControlECC* приведен на рис. 5.5, а его содержимое – на рис. 5.6. Данная модель составлена в соответствии с разделом 2.2.3 части 1 стандарта IEC 61499 [163]. Позиция $p1$ представляет состояние $s0$ автоматной модели интерпретатора *ECC*, позиции $p2$ и $p3$ – состояние $s1$. Переходы $t1$ и $t4$ сетевой модели соответствуют одноименным переходам в автоматной модели интерпретатора *ECC*, а переход $t3$ представляет цепочку $t2->s2->t3$ автоматной модели.

Приведенная сетевая модель функционирует следующим образом. При наличии сигнала вызова *ECC* (вход *invokeECC*), который порождается приходом какого-либо входного события на ФБ, при

нахождении интерпретатора *ECC* в начальном состоянии *s*0 (*m*(*p*1)=1) в операционную часть через выход *sample* выдается сигнал съема входных данных и интерпретатор *ECC* переходит в состояние *s*1 (*m*(*p*2)=1). Следует заметить, что съем данных в модели производится в том же шаге, что и выдача сигнала съема. После этих действий в операционную часть интерпретатора *ECC* через выход *eval_t* модуля *ControlECC* выдается сигнал для оценки разрешенности *EC*-переходов в текущем состоянии. Операционная часть интерпретатора *ECC* выбирает разрешенный *EC*-переход, осуществляет смену состояний интерпретируемой диаграммы *ECC* и выполняет все *EC*-акции, связанные с целевым состоянием данного *EC*-перехода. После этого в управляющую часть интерпретатора *ECC* выдается сигнал о завершении выполнения всех этих *EC*-акций (вход *acts_completed*) и интерпретатор *ECC* переходит в состояние *s*1 (*m*(*p*2)=1). При этом цикл по оценке и выполнению *EC*-переходов будет продолжен в дальнейшем. Если же в результате оценки разрешенности *EC*-переходов выясняется, что в диаграмме *ECC* нет разрешенных переходов, то операционная часть интерпретатора *ECC* выдает в управляющую часть соответствующий сигнал (вход *no_t_clears*). В этом случае интерпретатор *ECC* вновь переходит в начальное состояние *s*0 (*m*(*p*1)=1). Следует также заметить, что при приходе сигнала вызова *ECC* в случае, когда интерпретатор не находится в начальном состоянии (*m*(*p*1)=0), никаких действий не производится и сигнал теряется. При этом считается, что ФБ занят.

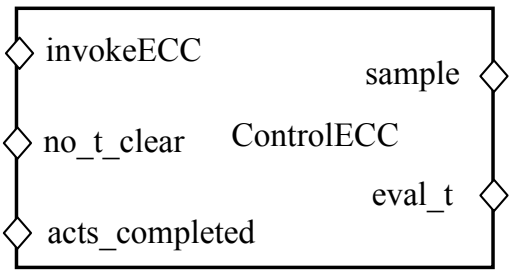


Рис. 5.5. Интерфейс *NCES*-модуля управления выполнением

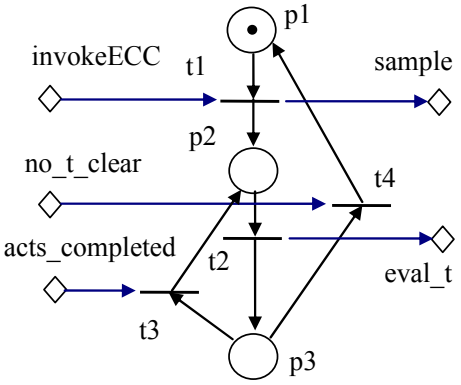


Рис. 5.6. Содержимое *NCES*-модуля управления выполнением

Моделирование условий *EC*-переходов

Шаблон *Cond*-модуля для вычисления условий переходов в диаграмме *ECC* (*EC*-переходов) включает входы-выходы *start*, *true* и *false*, а также (возможно) условный вход *ei* и арифметические вхо-

ды. По сигналу, поданному на событийный вход *start*, модуль начинает вычисления условия. По завершению вычислений сигнал подается на выход *true*, если условие истинно, и на выход *false*, если ложно. Необязательный условный вход *ei* используется, если в условии фигурирует событие. При этом данный вход связывается с внешней позицией, представляющей входную событийную переменную (*EI*-переменную). Арифметические входы используются, если в условии фигурируют переменные ФБ. С этими входами могут быть связаны внешние позиции, представляющие входные, выходные и внутренние переменные базисного ФБ.

На рис. 5.7 и 5.8 в качестве примера изображен один из вариантов *Cond*-модуля, вычисляющего условие *EC*-перехода *ei*1&(*di*2>10) в диаграмме *ECC*, представленной на рис. 5.11. Для разрешения конфликта между переходами *t*1 и *t*2 используется ингибиторная дуга. Переход *t*6 реализует функцию сравнения «Больше». Результат сравнения (0 или 1) записывается в позицию-переменную *p*3.

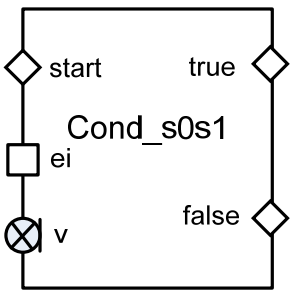


Рис. 5.7. Интерфейс *Cond*-модуля

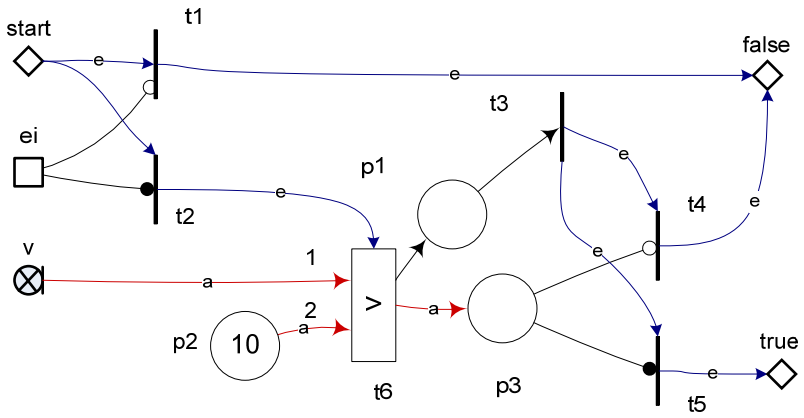


Рис. 5.8. Содержимое *Cond*-модуля

Моделирование алгоритмов

Алгоритмы в модели базисного ФБ так же, как и условия *EC*-переходов, представляются в виде модулей (называемых *Alg*-модулями). Шаблон *Alg*-модуля имеет один событийный вход *start*, событийный выход *end*, а также возможно несколько арифметических входов и выходов. *aNCES*-сети позволяют представлять алгоритмы с использованием двух подходов: 1) с явным выделением управляющей и операционной частей для пошагового выполнения алгоритма; 2) с использованием принципа потока данных, в соответствии с которым выполнение операций определяется готовно-

стью операндов.

На рис. 5.9 и 5.10 приведен пример *Alg*-модуля для моделирования простейшего алгоритма: *if* (*v*1>5) *then* v2:=*v*2+1; *else* *v*3:=3 с использованием первого подхода.

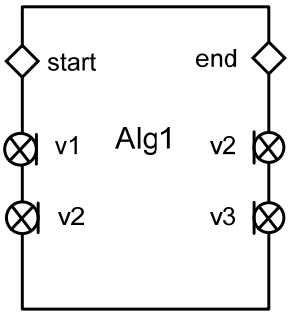


Рис. 5.9. Интерфейс *Alg*-модуля

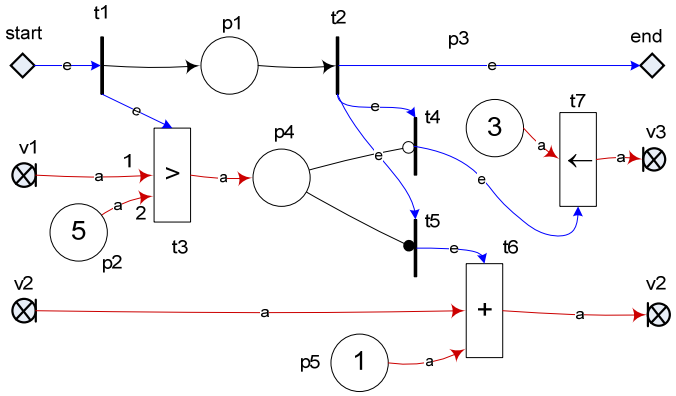


Рис. 5.10. Содержимое *Alg*-модуля

Моделирование диаграммы *ECC*

При моделировании диаграммы управления выполнением (диаграммы *ECC*) базисного ФБ следует выделять два момента: 1) собственно моделирование диаграммы *ECC*; 2) встраивание модели диаграммы *ECC* в окружение. При формировании сетевой модели диаграммы *ECC* используются следующие правила:

1) каждому *EC*-состоянию ставится в соответствие позиция сетевой модели. Начальному *EC*-состоянию соответствует маркированная позиция;

2) каждому *EC*-переходу ставится в соответствие переход сетевой модели. Входной позицией перехода становится позиция, представляющая исходное *EC*-состояние, а выходной позицией – позиция, представляющая целевое *EC*-состояние данного *EC*-перехода;

3) для каждой позиции-состояния создается переход сетевой модели (проверочный переход), причем данная позиция связывается с этим переходом условной дугой. Этот переход служит для определения того, находится ли диаграмма *ECC* в данном состоянии или нет.

На рис. 5.11 в качестве примера представлена диаграмма *ECC*, а на рис. 5.12 – ее сетевая модель (в виде модуля). С событийного входа *eval* поступает запрос для идентификации текущего состояния диаграммы *ECC*. Сигнал на событийном выходе out_i ($i \in 1...4$) свидетельствует о том, что *ECC* находится в состоянии s_i. Остальные событийные входы предназначены для смены соответствующих со-

стояний *ECC*. На рис. 5.13 показаны модуль диаграммы *ECC*, представленный на рис. 5.11, и сопутствующая ему инфраструктура.

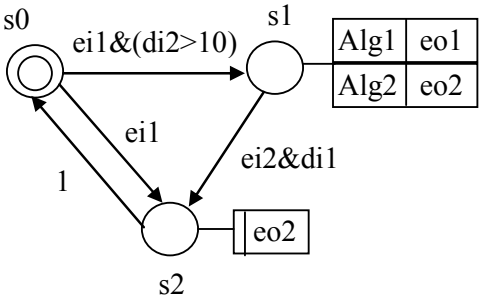


Рис. 5.11. Диаграмма *ECC*

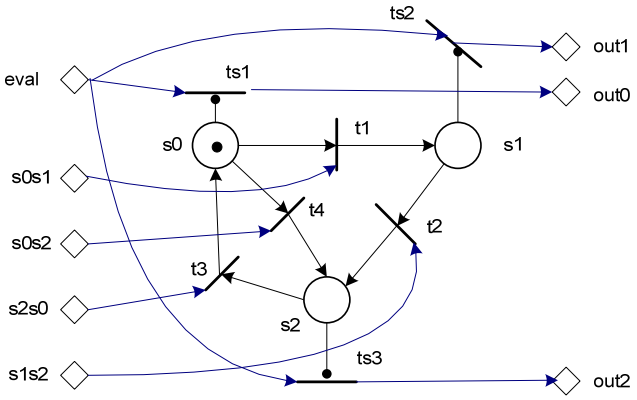


Рис. 5.12. Сетевая модель диаграммы *ECC*, представленной на рис. 5.11

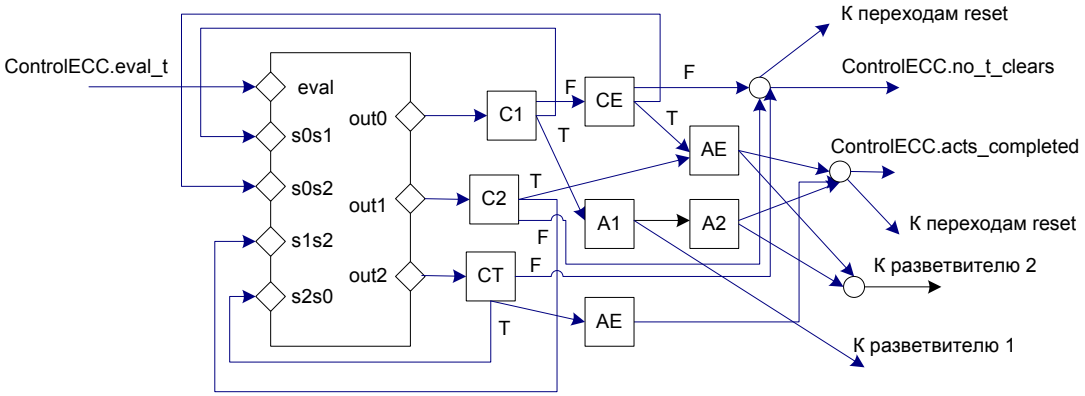


Рис. 5.13. Встраивание модели диаграммы *ECC*, приведенной на рис. 5.11, в окружение

Модули *C*1, *C*2, *CE*, *CT* являются *Cond*-модулями. Причем модуль *C*1 соответствует модулю *Cond_s*0*s*1, описанному выше (см. рис. 5.7, 5.8), модуль *CE* вычисляет условие, содержащее только событие, а модуль *CT* представляет тождественно истинное условие. Последний тип модуля в действительности не является необходимым, но он используется для унификации процедуры синтеза. На рис. 5.13 истинный выход *Cond*-модулей отмечен буквой *T*, а ложный – *F*. Следует обратить внимание, что модули *C*1 и *CE* образуют цепочку Дейзи. Модули *A*1 и *A*2 представляют соответственно *Alg*-модули для вычисления алгоритмов *Alg*1 и *Alg*2, отмеченных на *ECC* выше (в соответствии с рис. 5.11). Модуль *AE* определяет пустой алгоритм.

Кратко рассмотрим функционирование представленной систе-

мы. Допустим, что *ECC* находится в состоянии *s*0. При приходе сигнала оценки *EC*-переходов из модуля *ControlECC* (через вход *eval*) в модуле *ECC* вырабатывается сигнал *out*0, поступающий на вход *start Cond*-модуля *C*1. Если вычисленное условие окажется истинным, то выдается сигнал на вход *s*0*s*1 модуля *ECC* и моделируется переход из состояния *s*0 в состояние *s*1. Кроме того, этот сигнал поступает на вход *start* модуля алгоритма *A*1. После завершения его работы выдается сигнал на *Разветвитель* 1, осуществляющий выдачу выходного сигнала за пределы блока, и инициируется работа модуля алгоритма *Alg*2. После его завершения выдается сигнал: 1) на *Разветвитель* 2; 2) в модуль *ControlECC*, что свидетельствует о завершении выполнения всех акций в *EC*-состоянии; 3) на переходы типа *reset*, производящие сброс входных событийных переменных.

Если вычисленное в модуле *C*1 условие является ложным, то инициируется работа следующего *Cond*-модуля в цепочке Дейзи, а именно: модуля *CE*. Если результат вычисления и этого модуля также окажется ложным, то сигналы подаются: 1) в модуль *ControlECC* на вход *no_t_clears*, что свидетельствует, что в *EC*-состоянии *s*0 нет разрешенных *EC*-переходов; 2) на переходы *reset*.

Моделирование сети клапанов данных

Идея клапана данных (КД) и переход к одноуровневой («плоской») структуре систем ФБ были подробно рассмотрены в разделе 2.

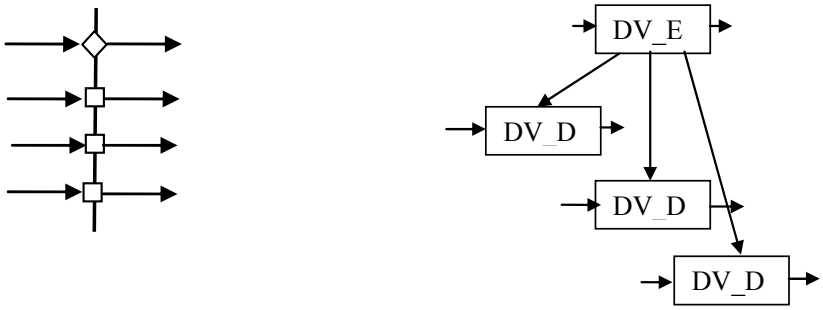


Рис. 5.14. Представления клапана данных

В общем случае каждый КД имеет один событийный вход-выход и несколько информационных входов-выходов. Для представления КД могут использоваться различные нотации (рис. 5.14). Сети КД используются для моделирования систем интерфейсов ФБ и их преставления в одноуровневой структуре ФБ [117]. Если КД представляет входной интерфейс ФБ, то назовем его входным, а если выходной – то выходным.

Моделирование сети КД с помощью сетевых моделей, подоб-

ных сетям Петри, но с произвольным синхронно-асинхронным срабатыванием переходов, подробно рассмотрено в [117]. В данном же разделе предлагается моделирование сетей КД с помощью *aNCES*-сетей. Это в известной степени сужает выбор модели выполнения сетей КД (до модели с асинхронным выполнением КД). Ниже приводятся показательные примеры построения моделей. Линейная структура из двух КД на рис. 5.15 представляется *aNCES*-сетью на рис. 5.16. Переходы типа *dve* моделируют разветвители, переходы типа *dvd* – копировщики, позиции типа *be* – буфера сигналов, позиции типа *bd* – буфера данных.

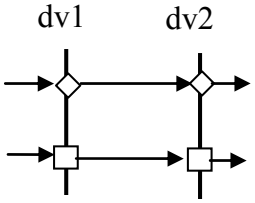


Рис. 5.15. Линейная структура из двух КД

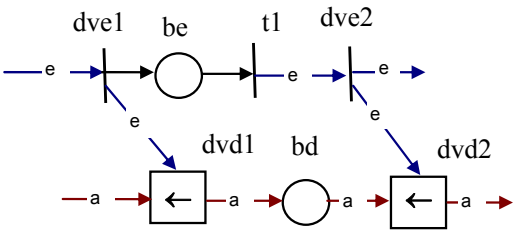


Рис. 5.16. Сетевая модель структуры, приведенной на рис. 5.15

Сеть КД, включающая конфигурацию типа соединитель и развилка для разветвителя сигналов, представлена на рис. 5.17. Сетевая модель этой сети КД приведена на рис. 5.18. Переход *dve*3 имеет ИЛИ-логику срабатывания.

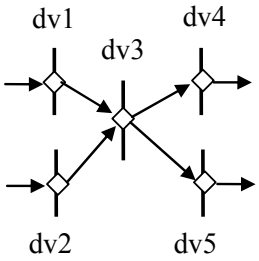


Рис. 5.17. Сеть КД, включающая соединитель и развилку сигналов

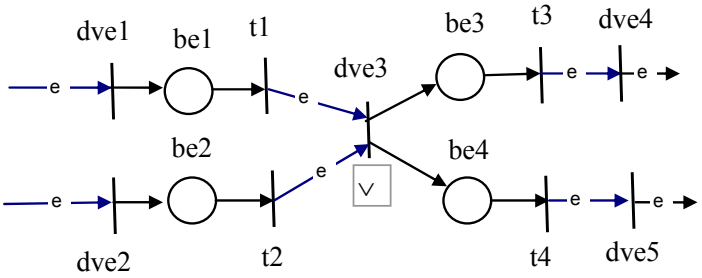


Рис. 5.18. Сетевая модель структуры, приведенной на рис. 5.17

Сеть КД, включающая конфигурацию типа развилка для копировщика, представлена на рис. 5.19. Сетевая модель этой сети КД приведена на рис. 5.20.

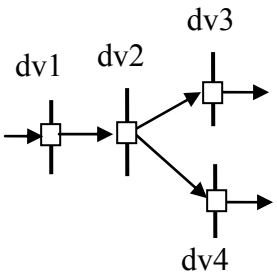


Рис. 5.19. Сеть КД, включающая развилку данных

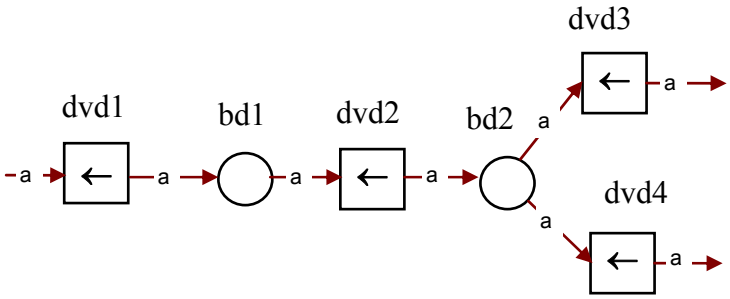


Рис. 5.20. Сетевая модель структуры, приведенной на рис. 5.19

Моделирование приема и выдачи сигналов и данных в базисных функциональных блоках

Системы интерфейсов составных ФБ моделируются с помощью сетей КД [25, 117]. Процессы, протекающие в интерфейсах базисных ФБ, более сложны и имеют свои особенности, поскольку блоки этого сорта являются неделимыми элементами системы ФБ. Проиллюстрируем моделирование цепей приема сигналов и данных в базисных ФБ на примере, представленном на рис. 5.21 и 5.22.

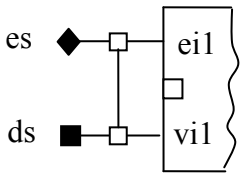


Рис. 5.21. Входной интерфейс базисного ФБ

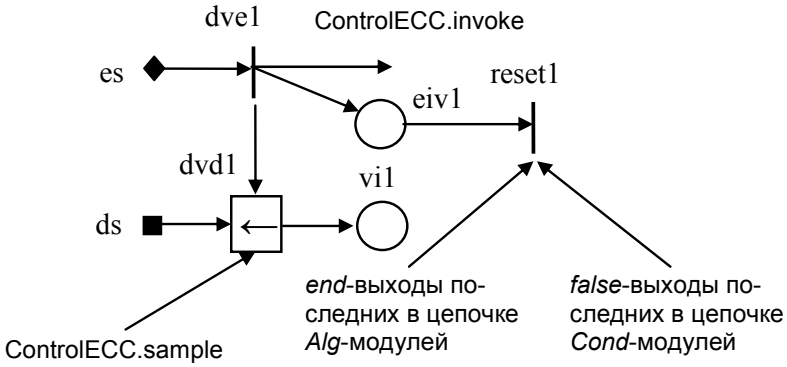


Рис. 5.22. Сетевая модель процессов, протекающих во входном интерфейсе, приведенном на рис. 5.21

Под *es* подразумевается некоторый источник события, а под *ds* – источник данных. Позиция *eiv*1 представляет входную событийную переменную (*EI*-переменную). Следует заметить, что в проекте стандарта IEC 61499 она существовала, однако впоследствии в окончательной версии стандарта она не упоминается. Несмотря на непрекращающиеся до сих пор споры о ее необходимости, в модели систем ФБ без нее трудно обойтись. *EI*-переменная устанавливается соответствующим приходящим входным событием (через разветви-

тель *dve*1) и сбрасывается переходом *reset*1 или после обработки всех *EC*-акций в каком-либо *EC*-состоянии, или в случае неудачной оценки *EC*-переходов. Эти предположения делают *EI*-переменную «короткоживущей», действительной максимум на срабатывание только одного *EC*-перехода.

Съем данных с информационного входа *vi*1 во входную переменную ФБ (позиция *vi*1) (см. рис. 5.22) производится только при наличии входного сигнала на событийном входе *ei*1 и нахождении интерпретатора *ECC* в незанятом состоянии (состоянии *s*0). В примере это (конъюнктивное) условие моделируется тем, что в переход-копировщик *dvd*1 входят две событийные дуги: одна с перехода-разветвителя сигналов *dve*1, а другая – из выхода *sample* модуля *ControlECC*.

Выдача сигналов и данных через интерфейс базисного ФБ проще, чем прием. На рис. 5.23 и рис. 5.24 в качестве примера рассмотрен простой выходной интерфейс базисного ФБ и процессы, протекающие в нем.

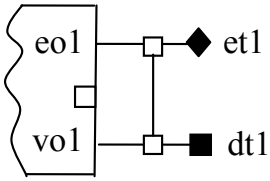


Рис. 5.23. Выходной интерфейс базисного ФБ

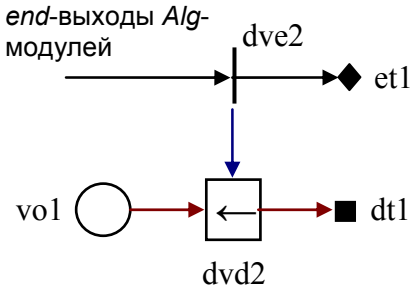


Рис. 5.24. Сетевая модель процессов, протекающих в выходном интерфейсе, приведенном на рис. 5.23

Под *et*1 понимается потребитель сигналов, а под *dt*1 – потребитель данных. Передача сигнала за пределы ФБ имитируется переходом *dve*2. Одновременно с передачей сигнала производится выдача наружу и данных из выходной переменной *vo*1. Работа перехода *dve*2 инициируется сигналами, отмечающими окончание работы соответствующих *Alg*-модулей. Следует заметить, что выдача сигналов и данных из ФБ происходит одновременно со сменой состояния ФБ. Это достигается использованием синхронного способа срабатывания переходов в *NCES*-сетях.

5.4. Метамодели исходной и целевой модельных форм

5.4.1. Метамодель систем функциональных блоков

Ниже определяется визуальный язык функциональных блоков, основанный на их представлении в виде типизированных атрибутных графов (ТАГ) [125]. Данный визуальный язык основан на стандарте IEC 61499 [24]. Мотивом разработки языка является создание языковой базы для трансформации описаний систем ФБ в другие модельные формы (в частности, в сетевые модели для проведения верификации системы) в среде трансформации графов *AGG*. Кроме «метамодельного» определения начального языка ФБ, в данном разделе представлены промежуточные метамодели, определяющие системы ФБ разных уровней, используемые в процессе синтеза формальных моделей.

Произведем классификацию представлений систем ФБ на основе степени полноты генерации одноуровневого представления. Будем выделять системы уровней 0, 1 и 2. Системами ФБ *уровня* 0 являются системы, в которых не раскрыт ни один (ссылочный экземпляр) ФБ. Система ФБ уровня 0 представляет исходную многоуровневую систему ФБ в виде иерархической типизированной структуры в начальной форме. В системах *уровня* 1 раскрыты все составные ФБ, базисные ФБ не раскрываются. Система ФБ *уровня* 2 представляет одноуровневую систему ФБ, в которой раскрыты все блоки, включая базисные. Эти системы состоят только из «начинок» базисных ФБ, связанных сетью клапанов данных (КД). Эту форму представления системы ФБ можно считать нормализованной, поскольку ее дальнейшее раскрытие невозможно. Системы уровня 1 можно считать промежуточной формой.

Следует заметить, что причиной «нераскрытия» некоторого ФБ может являться отсутствие описания соответствующего типа ФБ. Например, при проектировании «сверху-вниз» некоторые (типы) ФБ на текущей стадии проектирования могут быть еще не разработаны. Промежуточное представление системы ФБ (аналог сентенциальной формы в формальных грамматиках), тем не менее, может быть получено и использовано в будущем для полной генерации одноуровневого представления. К введенному выше признаку классификации систем ФБ добавим признак, определяющий, является ли система ФБ замкнутой или разомкнутой. Будем считать, что замкнутая система ФБ не имеет входов и выходов.

Приводимые ниже метамодели систем ФБ достаточно точны в определении структуры моделей, но в некоторых случаях допускают создание не вполне корректных (с точки зрения стандарта [163])

моделей. Это определяется следующим: 1) желанием уменьшить структурную сложность метамодели. Примером может служить рассмотрение алгоритмов только на уровне интерфейсов. При этом считается, что детальная работа с алгоритмами не входит в область компетенции данной метамодели; 2) определенной ограниченностью типизированных графов для метамоделирования. Для примера можно привести случай с кардинальностью связей, описываемый ниже; 3) вспомогательной ролью метамоделей в процессе трансформации графов. В системе *AGG* метамоделирование является одним из средств проверки и контроля производимых преобразований. Следует заметить, что дополнительные синтаксические и семантические ограничения, накладываемые на модель, могут быть выражены с помощью правил перезаписи графов.

На рис. 5.25 представлена метамодель базисного ФБ в виде типизированного графа. В данной метамодели используются следующие типы вершин: *CEI* и *CEO* – событийные вход и выход оболочки ФБ соответственно; *CDI* и *CDO* – информационные вход и выход оболочки ФБ соответственно; *S* – *EC*-состояние; *Cond* – условие *EC*-перехода; *Act* – *EC*-акция; *Var* – внутренняя переменная.

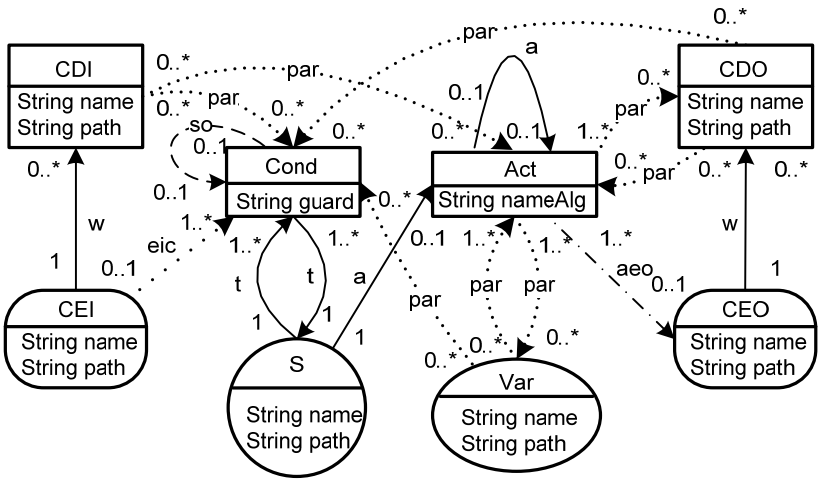


Рис. 5.25. Метамодель базисного ФБ

В метамодели базисного ФБ используются следующие типы дуг: *w* – *WITH*-связь; *eic* – дуга, связывающая событийный вход с условием *EC*-перехода; *aeo* – дуга, связывающая *EC*-акцию с событийным выходом; *t* – дуга, связывающая *EC*-состояния и *EC*-условия; *a* – дуга, связывающая *EC*-состояние с первой *EC*-акцией или *EC*-акции между собой согласно порядку их выполнения; *so* – дуга, определяющая порядок оценки *EC*-переходов, выходящих из одного *EC*-состояния; *par* – дуга для представления потоков данных через алгоритмы. Данная дуга служит также для определения

параметров условия *EC*-перехода.

Из всех используемых атрибутов вершин требует пояснения только атрибут *path* для входов-выходов оболочки. Данный атрибут представляет путь до родительского ссылочного экземпляра в дереве иерархии. Путь представляет собой конкатенацию имен экземпляров ФБ, лежащих на этом пути. Для более удобного его представления может использоваться точечная нотация. Атрибут *path* используется в правилах перехода к одноуровневому представлению. В предложенной метамодели базисного ФБ алгоритмы, связанные с *EC*-акцией, определяются только на уровне интерфейсов (входов-выходов). Для определения входных и выходных параметров алгоритма используются дуги типа *par*, входящие и выходящие из соответствующего элемента типа *Act*.

На рис. 5.26 приведена метамодель составного ФБ (и субприложения) в виде типизированного графа. Штрих-пунктирные линии для связей типов *ec* и *dc* относятся только к субприложению, а связи типа *w* относятся только к составному ФБ.

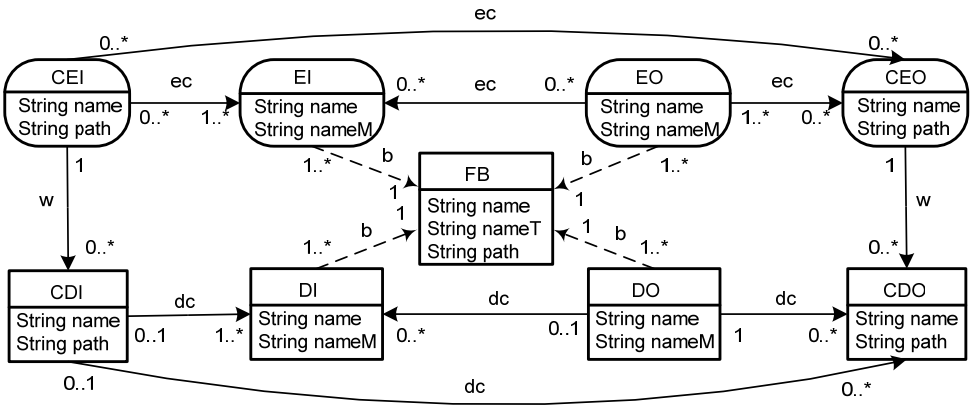


Рис. 5.26. Метамодель составного ФБ и субприложения

Как видно из рис. 5.26, метамодель составного ФБ существенно отличается от метамодели базисного ФБ. Общими являются лишь типы вершин, определяющие интерфейс ФБ (вершины типов *CEI*, *CEO*, *CDI*, *CDO*). Вершина типа *FB* представляет ссылочный экземпляр ФБ (субприложения), а вершины типов *EI*, *EO*, *DI*, *DO* – его интерфейс, причем вершина типа *EI* представляет событийный вход, вершина типа *EO* – событийный выход, вершина типа *DI* – информационный вход, а вершина типа *DO* – информационный выход. Дуги типа *b* определяют принадлежность данных интерфейсных вершин определенному ссылочному экземпляру ФБ (субприложения). Для представления потока событий используются дуги

типа *ec*, а для представления потока данных – дуги типа *dc*.

Кардинальности связей определяют допустимую структуру взаимосвязей экземпляров вершин достаточно точно, однако в ряде случаев этого недостаточно. Например, условие стандарта IEC 61499 (пункт 2.4.1), согласно которому к информационному входу экземпляра может быть подключено не более одной информационной линии, в данной модели не может быть выражено. Согласно метамодели на рис. 5.26 к информационному входу может быть подключен один информационный выход другого ФБ и один информационный вход оболочки (т.е. две информационные линии). Вершина типа *FB* имеет атрибут *name* – (локальное) имя экземпляра и атрибут *nameT* – имя типа ФБ. Атрибут *nameM* у интерфейсных типов вершин определяет иерархическое имя экземпляра ФБ-владельца.

Метамодель разомкнутых систем ФБ уровня 1 в виде типизированного графа представлена на рис. 5.27. Данная метамодель получается путем расширения метамодели составного ФБ элементами *DV_E* и *DV_D*, представляющими клапаны данных (КД), а также соответствующими связями. Тип вершины *DV_E* определяет развветвитель КД, а *DV_D* – копировщик КД.

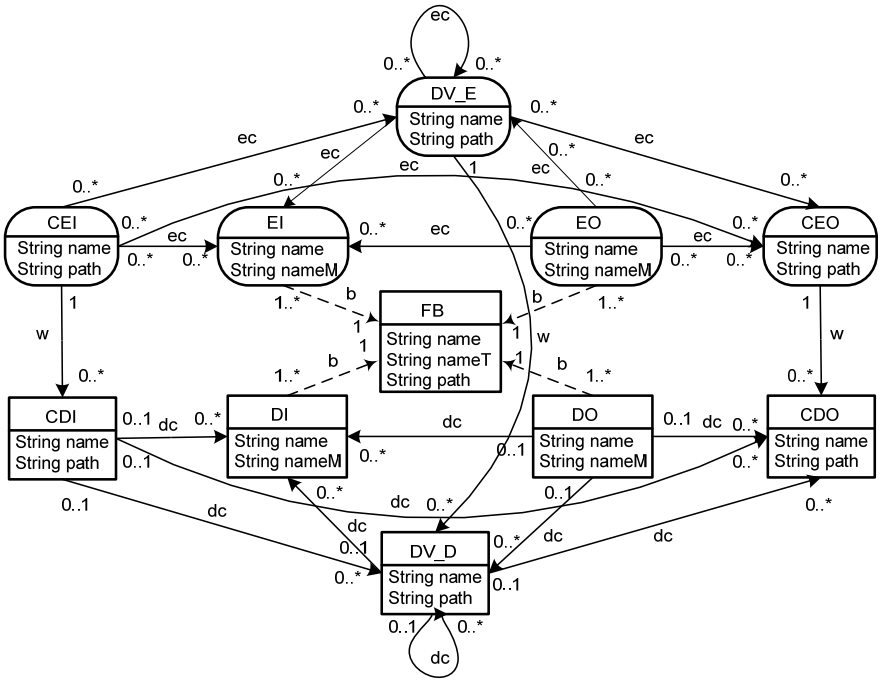


Рис. 5.27. Метамодель разомкнутых систем ФБ уровня 1

Метамодель замкнутых систем ФБ уровня 2 в виде типизированного графа представлена на рис. 5.28.

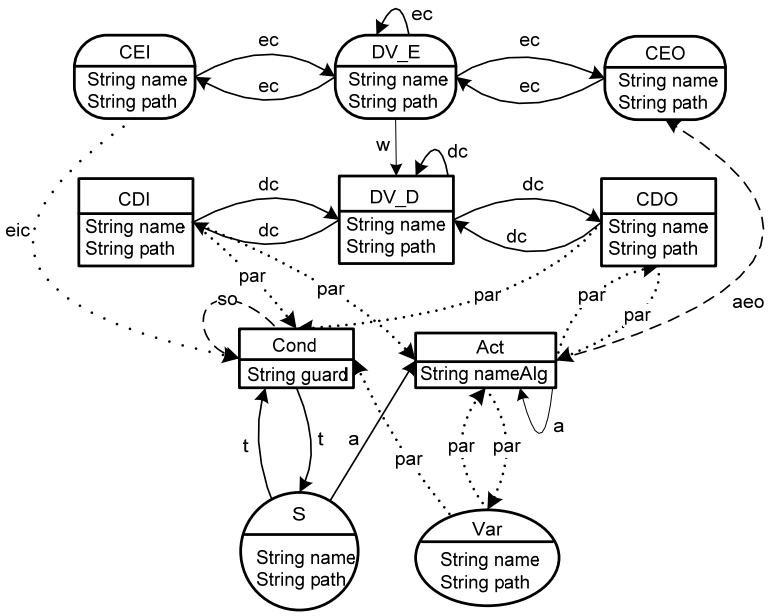


Рис. 5.28. Метамодель замкнутых систем ФБ уровня 2

Данная метамодель состоит из элементов, используемых в других метамоделях. Следует также отметить, что в процессе генерации одноуровневого представления системы ФБ, кроме отмеченных выше элементов, используются следующие виды вершин: вершина *NumM* (представляет счетчик модулей) и вершина *ParentType* (используется для хранения имени типа ФБ, порождающего экземпляр ФБ).

5.4.2. Метамодель арифметических *NCES*-сетей

В данном разделе представлен визуальный язык модульных *aNCES*-сетей, основанный на их представлении в виде типизированных атрибутных графов (ТАГ). Классификацию представлений сетевых моделей (*aNCES*-сетей) по степени полноты генерации одноуровневого эквивалента можно произвести аналогично тому, как это делалось для систем ФБ. Однако в целях простоты будем классифицировать сети на многоуровневые (уровень 0) и одноуровневые (уровень 1). Последние характеризуются тем, что в них раскрыты все модули. Основными элементами сети в этом случае являются только переходы и позиции. Для каждого уровня представления сетевой модели может быть разработана собственная метамодель.

На рис. 5.29 представлена метамодель модуля (уровня 0) многоуровневой доменно-ориентированной *aNCES*-сети в виде графа типов с учетом упрощений, принятых в [19]. К упрощениям *aNCES*-сети относится использование: 1) только одного типа «арифметического» перехода, моделирующего КД; 2) только трех

типов входов-выходов (из возможных семи) для определения интерфейса модуля. Для «облегчения» графического представления метамодели на рис. 5.29 не показаны атрибуты и кардинальности вершин и дуг.

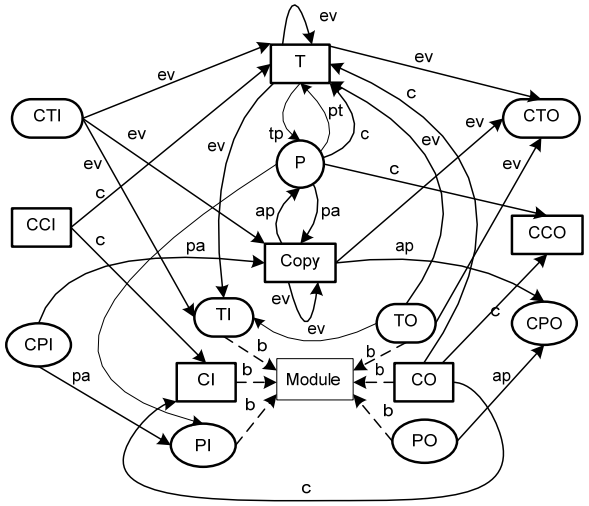
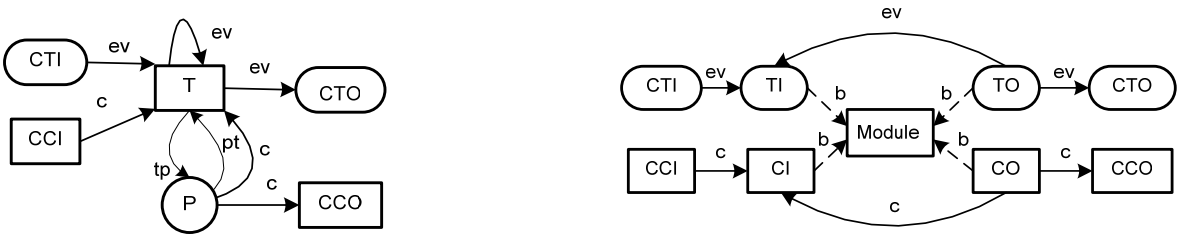


Рис. 5.29. Обобщенная метамодель модуля многоуровневой *aNCES*-сети

Вершины типов *CTI*, *CCI*, *CPI*, *CTO*, *CCO* и *CPO* определяют интерфейс модуля (сети), причем вершина типа *CTI* (*CTO*) представляет событийный вход (выход), *CCI* (*CCO*) – условный вход (выход), *CPI* (*CPO*) – вход (выход) по данным. Вершины типов *TI*, *CI*, *PI*, *TO*, *CO* и *PO* аналогичны перечисленным вершинам, но они определяют интерфейс ссылочного экземпляра модуля. Дуги типа *b* определяют принадлежность данных интерфейсных вершин определенному ссылочному экземпляру модуля. Вершина типа *T* представляет обычный переход сетевой модели, а вершина типа *Copy* – переход, моделирующий КД. В дальнейшем переход последнего типа будем также обозначать знаком стрелки ← (вместо слова *Copy*). Вершина типа *P* соответствует позиции сетевой модели, а вершина типа *Module* – ссылочному экземпляру модуля. Дуги типа *pt* и *tp* определяют связи между позициями и переходами и переходами и позициями соответственно. Связь типа *ev* служит для определения событийных дуг, а связь типа *c* – условных дуг. Дуги типа *pa* и *ap* представляют так называемые «арифметические» связи.

Следует отметить, что на основе метамодели, приведенной на рис. 5.29, путем ее *сужения* могут быть построены метамодели модулей (более простых в структурном плане) обычных *NCES*-сетей [181].Сужение метамодели достигается удалением определенных структурных элементов или манипулированием кратностью вершин, входящих в метамодель. Например, чтобы из приведенной

метамодели получить метамодель базисного модуля *NCES*-сети достаточно положить кратность всех вершин, кроме вершин *CTI*, *CCI*, *CTO*, *CCO*, *T* и *P*, равной нулю. В качестве примера на рис. 5.30 представлены метамодели базисного и составного модулей *NCES*-сетей, получаемые из метамодели (см. рис. 5.29) путем удаления из нее элементов с нулевой кратностью. Для сжатости изображения в метамоделях, приведенных на рис. 5.30, не указаны атрибуты и кратности вершин и дуг.



а) *б*)

Рис. 5.30. Метамодели базисного (*а*)
и составного (*б*) модулей *NCES*-сетей

На рис. 5.31 представлена метамодель одноуровневой *aNCES*-сети (с отображением атрибутов и кардинальностей вершин и дуг), сформированная средствами системы *AGG*.

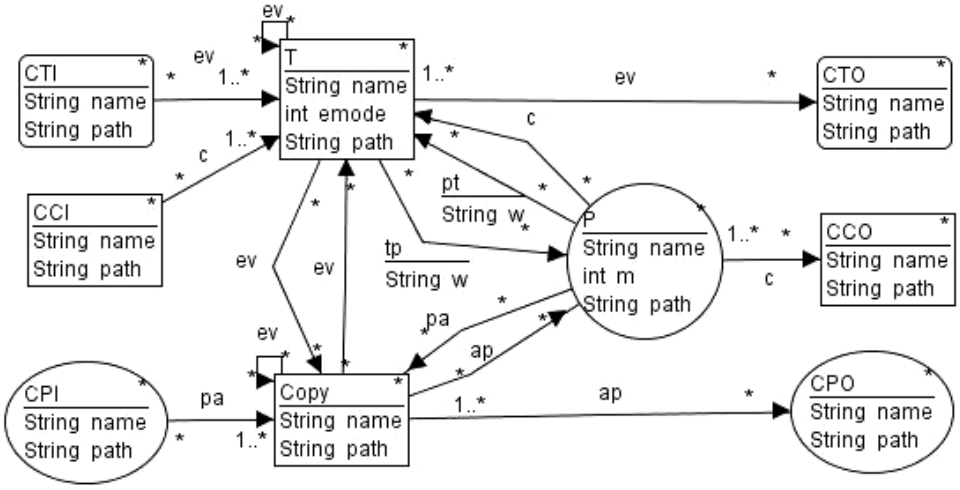


Рис. 5.31. Граф типов для одноуровневой *aNCES*-сети в системе *AGG*

На приведенном рисунке вершина *P* характеризуется атрибутом *m* (маркировка позиции). Вершина *T* имеет атрибут *emode* – режим срабатывания перехода (И-логика/ИЛИ-логика). Атрибут *path* во всех вершинах типизированного графа на рис. 5.31 имеет такой же смысл, что и одноименный атрибут в системах ФБ. Атрибут *name* определяет имя соответствующего элемента. Следует отметить, что

для перехода от многоуровневых к одноуровневым сетям используются следующие дополнительные (служебные) типы вершин, не отображенные в метамоделях: *NumP* – счетчик числа позиций, *NumT* – счетчик числа переходов, *NumM* – счетчик числа модулей.

5.5. Правила перехода от многоуровневой структуры систем функциональных блоков к одноуровневой структуре

Переход от многоуровневой структуры системы ФБ к одноуровневой структуре производится в два этапа. На первом этапе (этапе компоновки) рекурсивно осуществляется замена ссылочных экземпляров ФБ на соответствующие им развернутые экземпляры. На втором этапе (этапе встраивания) производится встраивание развернутых экземпляров ФБ в окружающую систему.

Шаблон правила замены ссылочного экземпляра составного ФБ на соответствующий развернутый экземпляр приведен на рис. 5.32.

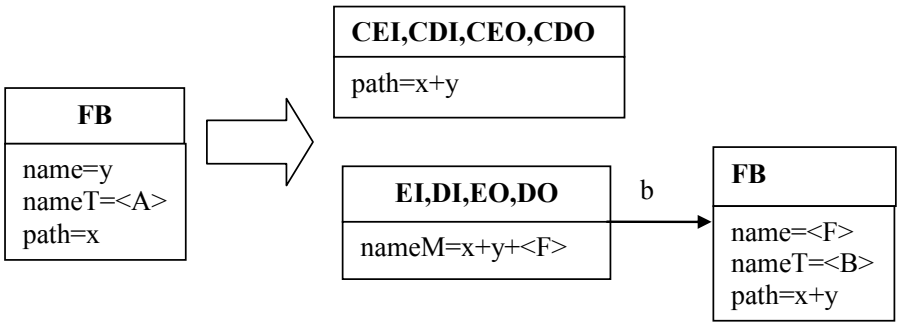


Рис. 5.32. Шаблон правила замены ссылочного экземпляра составного ФБ на соответствующий развернутый экземпляр

В левой части правила записывается вершина типа *FB*, представляющая конкретный тип ФБ. В правой части правила записывается содержимое этого типа ФБ в соответствии с метамоделью ФБ. Для интерфейсных вершин *CEI*, *CDI*, *CEO* и *CDO* правой части правила определяются выражения для вычисления ряда атрибутов. Это необходимо для процедуры встраивания развернутого экземпляра ФБ в генерируемую одноуровневую модель. На рис. 5.32 принято, что в правых частях атрибутных выражений переменные записываются с маленькой буквы, а константы (конкретные значения) указаны в виде больших букв в угловых скобках.

Пусть имеется ссылочный экземпляр ФБ с локальным именем *y* и иерархическим именем родительского экземпляра *x*. Тогда иерархическое имя соответствующего развернутого экземпляра образует-

ся как *x+y*, где + означает операции конкатенации (через точку). Это значение записывается в атрибут *path* для интерфейсных элементов оболочки, а также для вложенных элементов типа *FB* в правой части правила. Для интерфейсных элементов ссылочных экземпляров ФБ правой части правила вычисляется атрибут *nameM*, определяющий иерархическое имя экземпляра ФБ-владельца.

Шаблон правила замены ссылочного экземпляра базисного ФБ на соответствующий развернутый экземпляр представлен на рис. 5.33.

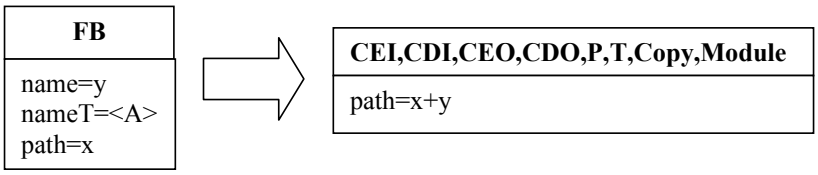


Рис. 5.33. Шаблон правила замены ссылочного экземпляра базисного ФБ на соответствующий развернутый экземпляр

Как видно из данного рисунка, практически для всех элементов правой части правила вычисляется атрибут *path*. Для неинтерфейсных элементов этот атрибут может использоваться в дальнейшем для идентификации конкретного элемента в сгенерированной иерархической модели системы ФБ.

Все правила перехода к одноуровневой структуре функционально можно разбить на несколько групп. Классификация правил приведена на рис. 5.34. Характеристика и примеры правил каждого класса приведены ниже.

Правила создания клапанов данных

Как видно из рис. 5.34, существуют четыре типа правил для создания клапанов данных (КД). В процессе развертывания системы ФБ клапаны данных создаются для одноименных входов-выходов ссылочного экземпляра ФБ и соответствующей ему оболочки развернутого экземпляра ФБ. Назовем эти одноименные входы-выходы сопряженными. Правила для создания КД являются однотипными, поэтому рассмотрим их на примере правила создания входного разветвителя *DV_E* (рис. 5.35).

Как видно из рисунка, правило применяется при выполнении следующих условий:

1) *EI.nameM = CEI.path*. Иными словами, иерархическое имя ссылочного экземпляра ФБ, которому принадлежит событийный вход, должно совпадать с именем развернутого экземпляра ФБ, которому принадлежит событийный вход оболочки;

2) *EI.name = CEI.name*. Иными словами, имя событийного входа ссылочного экземпляра ФБ должно совпадать с именем событийного входа оболочки развернутого экземпляра ФБ.

Рис. 5.34. Классификация правил перехода от многоуровневой структуры систем ФБ к одноуровневой структуре

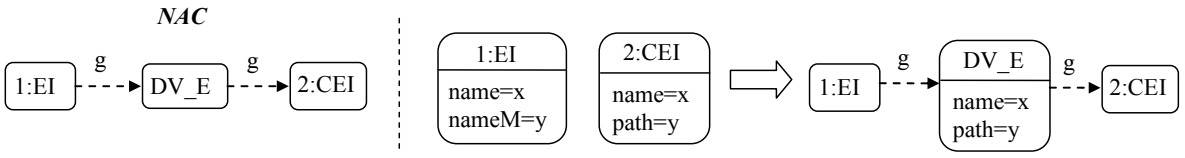


Рис. 5.35. Правило создания входного разветвителя КД

Созданный разветвитель включается в цепь между событийными входами типов *EI* и *CEI*, которые являются его родителями. Для связывания используется общая дуга типа g (*General*). Условие *NAC* предназначено для исключения создания дублирующих разветвителей.

Правила связывания клапанов данных

Суть правил связывания КД по событийным и информационным линиям заключается в том, что все связи, подходящие к сопря-

женным входам (выходам) ссылочного экземпляра ФБ и оболочки развернутого экземпляра ФБ, перебрасываются на соответствующий им клапан данных. Можно сказать, что КД наследует связи своих родителей.

Правила связывания КД однотипные. Общее число правил равно 22. Разнообразие правил определяется следующими факторами: 1) рассматривается разветвитель или копировщик; 2) рассматривается входной или выходной КД; 3) перебрасываются связи по входам или по выходам; 4) какой тип элемента является источником (приемником) рассматриваемой цепочки.

В качестве примера рассмотрим правило переброски связи для выходного копировщика (по выходам) (рис. 5.36). Источник связи между информационным выходом одного ссылочного экземпляра ФБ и информационным входом другого ссылочного экземпляра ФБ заменяется на выходной копировщик, являющийся дочерним по отношению к информационному выходу первого ссылочного экземпляра ФБ.

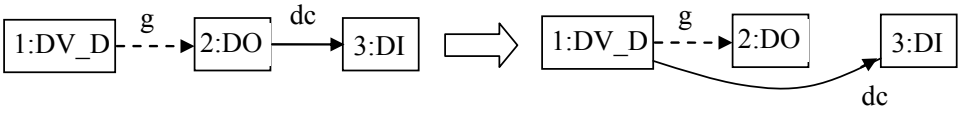


Рис. 5.36. Правило переброски связи для выходного копировщика по выходам

Существует два правила связывания КД по *WITH*-связям. Данные правила устанавливают *WITH*-связи между входными (выходными) разветвителями и копировщиками путем переброски *WITH*-связей между событийными и информационными входами (выходами) оболочки родительского развернутого экземпляра ФБ. Одно из правил (для входов) представлено на рис. 5.37.

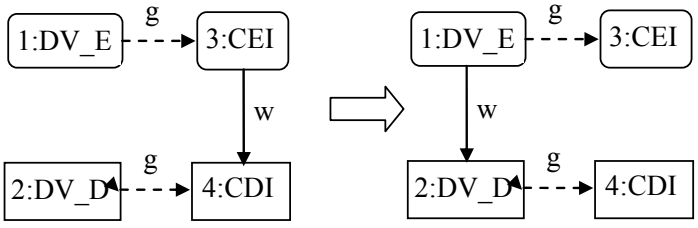


Рис. 5.37. Правило связывания входного клапана данных по *WITH*-связям

Правила удаления элементов

Правила удаления входов и выходов ссылочных экземпляров

ФБ удаляют те входы-выходы *EI*, *EO*, *DI* и *DO*, которые не принадлежат никаким ссылочным экземплярам ФБ. Если вход (выход) связан с элементом *FB* связью типа *b* (*belong to*), то этот вход (выход) не удаляется (условие *NAC*). На рис. 5.38 в качестве примера приведено правило удаления событийного входа *EI*.

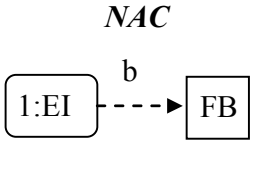


Рис. 5.38. Правило удаления событийного входа ФБ

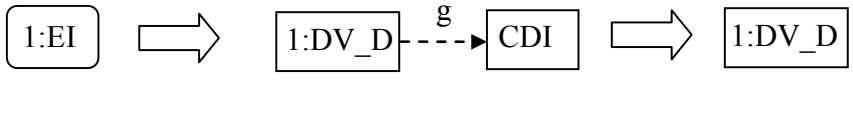


Рис. 5.39. Правило удаления информационного входа оболочки

Правила удаления входов-выходов оболочек (элементов *CEI*, *CEO*, *CDI*, *CDO*) удаляют только те входы-выходы оболочек, которые связаны с КД, т.е. входы-выходы оболочек развернутых экземпляров ФБ. Входы-выходы оболочки, не относящиеся к этому классу, относятся к самой внешней оболочке и не удаляются, поскольку они представляют интерфейс системы ФБ верхнего уровня. В качестве примера на рис. 5.39 приведено правило для удаления информационного входа оболочки.

5.6. Правила синтеза многоуровневых *aNCES*-сетей на основе одноуровневых систем функциональных блоков

Все правила из набора *R*2 можно разделить на следующие группы:

1) правила создания элементов;
2) правила связывания элементов;
3) правила создания и связывания элементов;
4) правила замены элементов;
5) правила установки атрибутов элементов;
6) правила удаления элементов.

Правила первого типа в чистом виде встречаются довольно редко. Обычно в правиле комбинируются действия по созданию и связыванию элементов.

Наиболее важные для синтеза правила приведены в табл. 5.1. Все перечисленные правила имеют один и тот же приоритет, за исключением правил удаления, приоритет которых ниже. Кроме пере-

численных в таблице правил, существуют правила удаления элементов *CEI*, *CEO*, *CDI*, *CDO*, *S*, *Cond*, *Act*, *Var*, *DV_E*, *DV_D*, *EI*, *EO*, *DI* и *DO*, а также правило создания счетчиков.

Таблица 5.1

Список правил синтеза многоуровневых *aNCES*-сетей на основе одноуровневых систем ФБ

Номер правила	Интерпретация правила
Правила создания и связывания элементов	
1	Создание перехода сетевой модели, моделирующего разветвитель КД
2	Создание арифметического перехода «Копирование», моделирующего копировщик КД, и позицию – буфер данных, связанную с этим переходом
3	Создание цепочки из элементов (позиций и переходов), моделирующую прохождение сигнала между соседними разветвителями
4	Связывание позиции-буфера данных одного перехода-копировщика с соседним переходом-копировщиком
5	Создание событийной связи между переходом, представляющим разветвитель, и арифметическим переходом, представляющим копировщик, на основе *WITH*-связи между разветвителем и копировщиком
6	Создание *NCES*-модуля типа *ControlECC*
7	Создание *Cond*-модуля для *EC*-перехода
8	Создание первого *Alg*-модуля цепочки модулей, прикрепленных к *EC*-состоянию
9	Создание непервого *Alg*-модуля цепочки модулей, прикрепленных к *EC*-состоянию и соединение выхода *end* предшествующего модуля с входом *start* вновь созданного модуля
10	Создание позиции, представляющей выходную переменную
11	Создание цепи для хранения и сброса событийного сигнала, включающей позицию типа *eiv* и переход типа *reset*
12	Создание позиции-состояния и проверочного перехода в модели *ECC*, связывание выхода *eval_t* модуля *ControlECC* с проверочным переходом
13	Создание перехода сетевой модели, имитирующего *EC*-переход, и его встраивание в *NCES*-модель *ECC*
14	Создание позиции сетевой модели, представляющей внутреннюю переменную ФБ

Продолжение табл. 5.1

Номер правила	Интерпретация правила

Правила связывания элементов	
15	Связывание выхода *sample* модуля *ControlECC* с входными копировщиками
16	Связывание проверочного перехода *NCES*-модели ECC с входом *start* первого *Cond*-модуля в цепочке, связанной с соответствующим *EC*-состоянием
17	Связывание двух *Cond*-модулей в цепочку, причем выход *false* первого модуля соединяется с входом *start* другого модуля
18	Соединение выхода *false* последнего *Cond*-модуля с входом *no_t_clears* модуля *ControlECC*
19	Соединение выхода *false* последнего *Cond*-модуля в цепочке с переходом типа *reset* (для сброса событийной переменной)
20	Соединение выхода *true Cond*-модуля с входом *start* первого *Alg*-модуля соответствующего *EC*-состояния
21	Соединение выхода *end* последнего *Alg*-модуля цепочки с переходом типа *reset* (для сброса событийной переменной)
22	Соединение выхода *end* последнего *Alg*-модуля цепочки с входом *acts_completed* модуля *ControlECC*
23	Связывание *Alg*-модуля с выходным разветвителем сигналов, а также установка ИЛИ-логики для разветвителя
24	Связывание позиции сетевой модели, представляющей выходную переменную, с копировщиком из окружения ФБ
25	Связывание входной событийной переменной (типа *eiv*) с *Cond*-модулем
26	Связывание входной переменной (позиции) ФБ с *Alg*-модулем
27	Связывание входной переменной (позиции) с *Cond*-модулем
28	Связывание *Alg*-модуля с позицией выходной переменной ФБ
29	Связывание позиции – выходной переменной ФБ с *Alg*-модулем
30	Связывание позиции – выходной переменной ФБ с *Cond*-модулем
31	Связывание позиции – внутренней переменной ФБ с *Alg*-модулем
32	Связывание позиции – внутренней переменной ФБ с *Cond*-модулем
33	Связывание *Alg*-модуля с позицией внутренней переменной ФБ
Правила замены элементов	
34	Преобразование внешнего событийного входа *CEI* системы ФБ во внешний событийный вход *CTI* модуля *NCES*
35	Преобразование внешнего событийного выхода *CEO* системы ФБ во внешний событийный выход *CTO* модуля *NCES*

Окончание табл. 5.1

36	Преобразование внешнего информационного входа *CDI* системы ФБ во внешний информационный вход *CPI* модуля *NCES*

37	Преобразование внешнего информационного выхода *CDO* системы ФБ во внешний информационный выход *CPO* модуля *NCES* (с удалением выходной позиции перехода *Copy*)
Правила установки атрибутов элементов	
38	Установка маркировки начальной *ECC*-позиции, равной 1
39	Установка имени типа *Alg*-модуля, соответствующего пустому алгоритму, равному *AlgNULL*
40	Установка имени типа *Cond*-модуля, соответствующему всегда истинному условию («1»), равному *CondTRUE*
41	Установка имени типа *Cond*-модуля, соответствующему условию, в котором присутствует только один входной событийный сигнал (без сторожевого условия), равному *CondEVENT*

Для примера рассмотрим правила из каждой группы.

Правило на рис. 5.40 создает доменно-ориентированный переход *aNCES*-сети «Копирование», моделирующий копировщик *DV_D*, и позицию – буфер данных, связанную с этим переходом. *NAC*-условие запрещает повторное применение этого правила в случае, если переход «Копирование» уже был ранее создан. Переход «Копирование» реализует управляемую операцию присваивания и помечается на рисунках словом «*Copy*» или стрелкой ←. На рис. 5.40 не показаны вершины типов *NumP* и *NumT*, определяющие *x* – текущий номер позиции и *y* – текущий номер перехода соответственно.

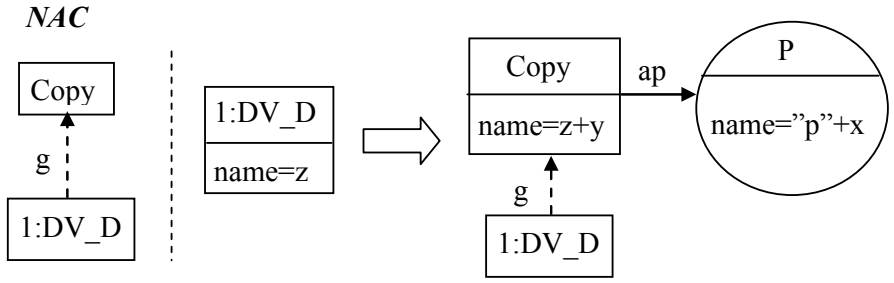


Рис. 5.40. Правило создания перехода-копировщика и позиции-буфера данных

Правило создания *Cond*-модуля для *EC*-перехода представлено на рис. 5.41.

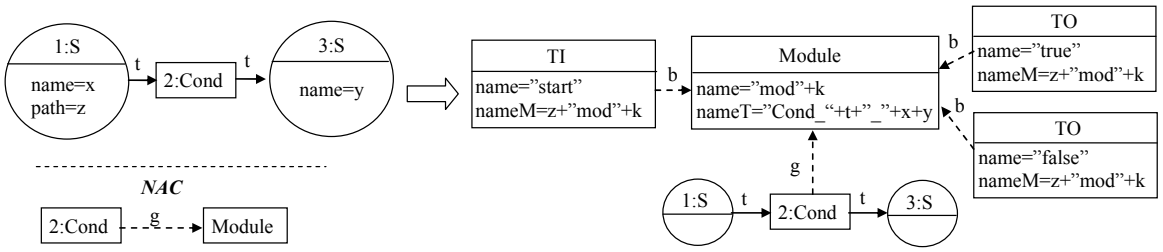


Рис. 5.41. Правило создания *Cond*-модуля

Данное правило создает для *Cond*-вершины модели системы ФБ соответствующий *Cond*-модуль сетевой модели, вычисляющий *EC*-условие, определенное в этой *Cond*-вершине. Модуль *Cond* создается вместе с входами-выходами, инвариантными для всех *Cond*-модулей, а именно: с входом *start* и выходами *true* и *false*. Соответствие *Cond*-вершины и *Cond*-модуля фиксируется с помощью дуги типа *g*. По определенным правилам формируется имя экземпляра и имя типа *Cond*-модуля. В образовании имени экземпляра участвует (сквозной) номер модуля *k* и имена *EC*-состояний (*x* и *y*), инцидентных *EC*-переходу, для которого создается данный модуль. В образовании имени типа модуля используется *t*-имя типа ФБ-родителя. Иерархическое родительское имя фиксируется в атрибуте *path*. В правиле на рис. 5.41 не показаны следующие вспомогательные вершины: вершина *NumM*, определяющая текущий номер модуля – *k* и вершина *ParentType*, хранящая тип родительского ФБ – *t*.

Связывание двух элементов сетевой модели может быть двух видов: простым – с генерацией только связи, более сложным – с созданием, кроме связи, дополнительных элементов. Пример связывания первого вида представлен правилом на рис. 5.42. Приведенное правило производит связывание *aNCES*-модуля, представляющего алгоритм, с выходным разветвителем сигналов. Правило можно интерпретировать следующим образом: «Если в системе ФБ имеется некоторая *EC*-акция (*Act*-вершина) и с этой *EC*-акцией связан (с помощью дуги *aeo*) выходной сигнал (вершина *CEO*), который передается некоторым потребителям (через разветвитель *DV_E*), причем уже создан *Alg*-модуль, соответствующий *Act*-вершине, и переход, моделирующий разветвитель, то необходимо соединить выход *end Alg*-модуля с этим переходом. Кроме того, устанавливается ИЛИ-логика для этого перехода, так как один и тот же выходной сигнал может выдаваться различными *EC*-акциями».

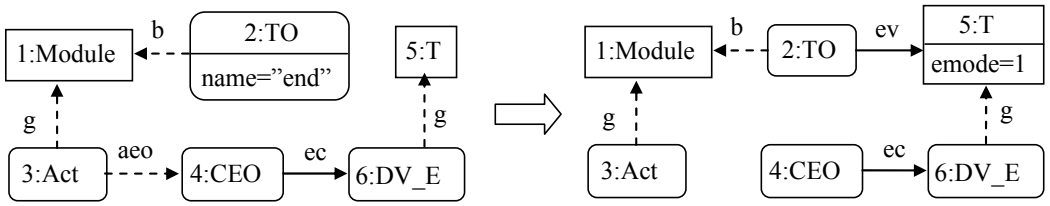


Рис. 5.42. Связывание *Alg*-модуля с выходным разветвителем сигналов

Пример связывания второго вида представлен правилом на рис. 5.43. Суть данного правила заключается в том, что если в условии *EC*-перехода (*Cond*-вершина) фигурирует некоторая входная переменная (вершина *CDI* и связь *par*), причем имеется реальный источник для этой переменной (копировщик *DV_D*) и, кроме того, уже создан переход типа *Copy* с выходной позицией, моделирующие этот копировщик вместе с соответствующим буфером данных, и создан *Cond*-модуль для упомянутой *Cond*-вершины, то необходимо связать позицию – буфер данных с *Cond*-модулем. Для этого создается арифметический вход *Cond*-модуля (элемент *PI*), который связывается арифметической дугой (типа *pa*) с позицией-буфером данных. Следует отметить, что под *Cond*-модулем понимается модуль *aNCES*-сети, моделирующий условие *EC*-перехода.

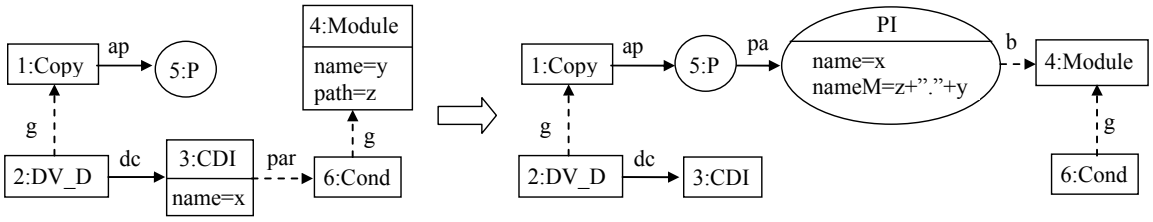


Рис. 5.43. Связывание входной переменной (позиции) с *Cond*-модулем

Существуют более сложные виды правил, комбинирующие как действия по созданию новых элементов, так и по связыванию вновь созданных и существующих элементов. В качестве примера такого правила рассмотрим правило на рис. 5.44. Данное правило создает переход сетевой модели, моделирующий *EC*-переход, встраивает данный переход сетевой модели в общую сетевую модель *ECC* и соединяет событийной дугой выход *true* соответствующего (уже существующего) *Cond*-модуля с созданным переходом. На рис. 5.44 переменная *x* определяет текущее значение счетчика переходов.

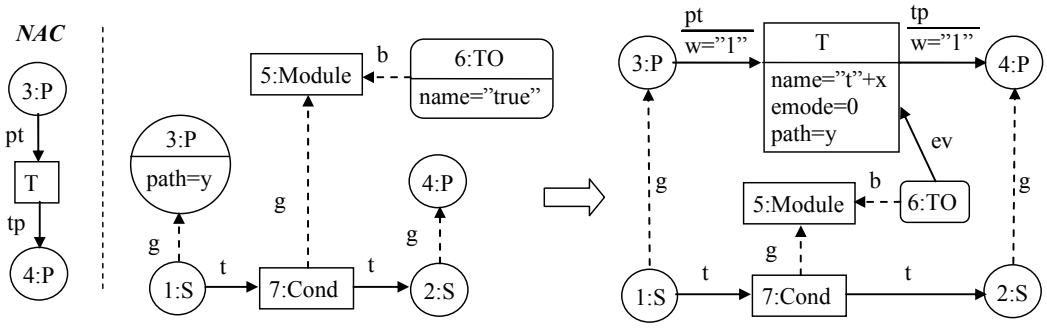


Рис. 5.44. Правило создания перехода сетевой модели, имитирующего *EC*-переход, и его встраивание в *aNCES*-модель *ECC*

Правило на рис. 5.45 устанавливает имя типа *Cond*-модуля, соответствующего условию (*Cond*-вершине), в котором присутствует только один входной событийный сигнал (без сторожевого условия), равному *CondEVENT*.

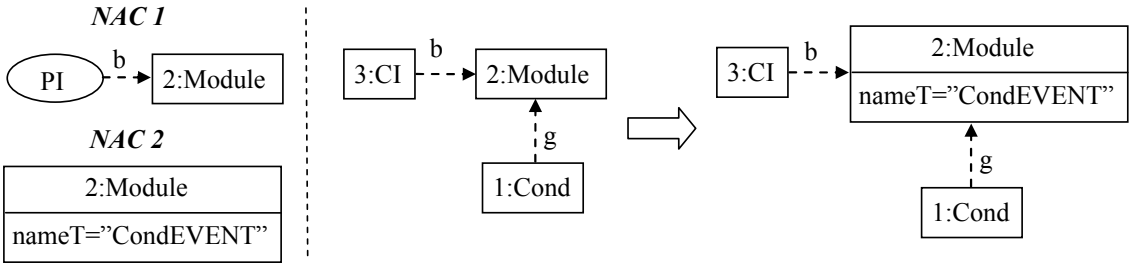


Рис. 5.45. Правило установки имени типа *Cond*-модуля, соответствующего *EC*-условию, в котором присутствует только входной событийный сигнал

Правила замены элементов преобразуют внешние входы-выходы системы ФБ во внешние входы-выходы сетевой модели. Например, имеется правило преобразования внешнего событийного входа *CEI* системы ФБ во внешний событийный вход *CTI* модуля *aNCES*. Правила удаления элементов очищают граф от элементов системы ФБ, оставляя при этом элементы сетевой модели.

В заключение следует отметить, что для задания порядка выполнения наборов правил используются приоритетные уровни (называемые в *AGG* слоями). Чем больше номер слоя, тем меньше приоритет. В проекте используются следующие слои: 0) правила подстановки содержимого ФБ вместо ссылочных экземпляров ФБ, правило создания вспомогательных вершин; 1) правила встраивания развернутых экземпляров ФБ в окружающую систему; 2) правило создания счетчиков переходов, позиций и модулей для *aNCES*-сетей; 3) правила синтеза многоуровневых *aNCES*-сетей на основе

одноуровневых систем ФБ; 4) правила удаления ненужных элементов системы ФБ; 5) правила подстановки содержимого *aNCES*-модулей вместо ссылочных экземпляров модулей; 6) правила встраивания развернутых экземпляров *aNCES*-модулей в окружающую систему; 7) правила удаления ненужных элементов *aNCES*-сети.

5.7 Пример. Синтез сетевой модели для функционального блока «RS-триггер»

В качестве примера рассмотрим синтез целочисленной *aNCES*-модели для простейшей системы, включающей один базисный ФБ типа *E_SR*, представляющий *RS*-триггер (рис. 5.46). Алгоритм *SET* устанавливает выходную булеву переменную в значение «Истина», а алгоритм *RESET* – в «Ложь».

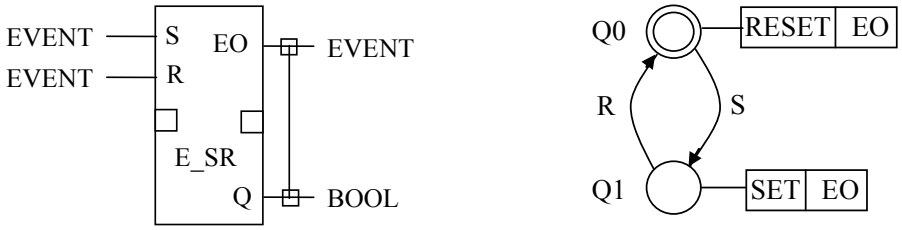


Рис. 5.46. Графическое представление ФБ *E_SR*:
интерфейс ФБ (слева) и диаграмма *ECC* (справа)

Представление ФБ *E_SR* в виде правила перезаписи графов в системе *AGG* приведено на рис. 5.47. Правая часть правила построена в соответствии с метамоделью базисного ФБ и с учетом положений о построении шаблонов правил компоновки.

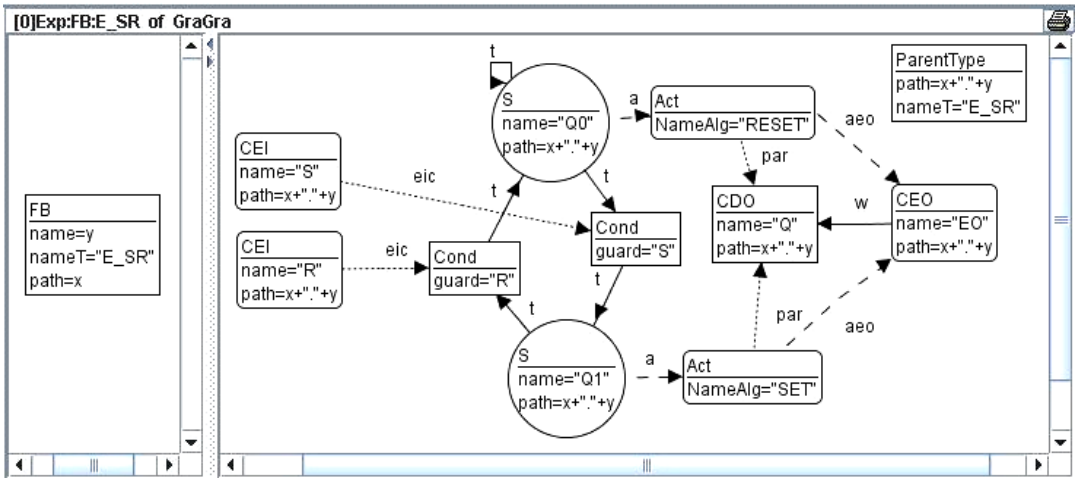


Рис. 5.47. Представление (типа) функционального блока *E_SR*
в виде правила перезаписи графов в системе *AGG*

Графовое представление простой системы функциональных блоков, включающей один ФБ *E_SR*, изображено на рис. 5.48. Эта система соответствует метамодели составных ФБ.

Ниже на рис. 5.49 приведено графовое представление модульной *aNCES*-сети, полученное в результате преобразований графовой модели системы ФБ, приведенной на рис. 5.47, 5.48, в системе *AGG*. Следует отметить, что сетевой фрагмент представлен в виде *aNCES*-модуля, имеющего входы-выходы.

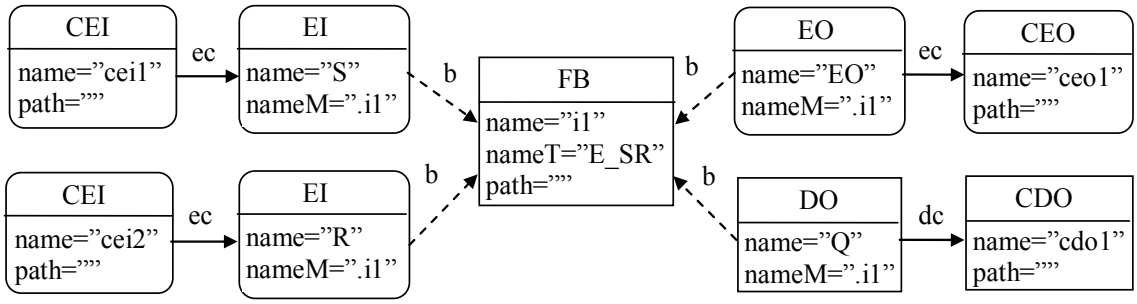


Рис. 5.48. Система ФБ,
включающая один функциональный блок *E_SR*

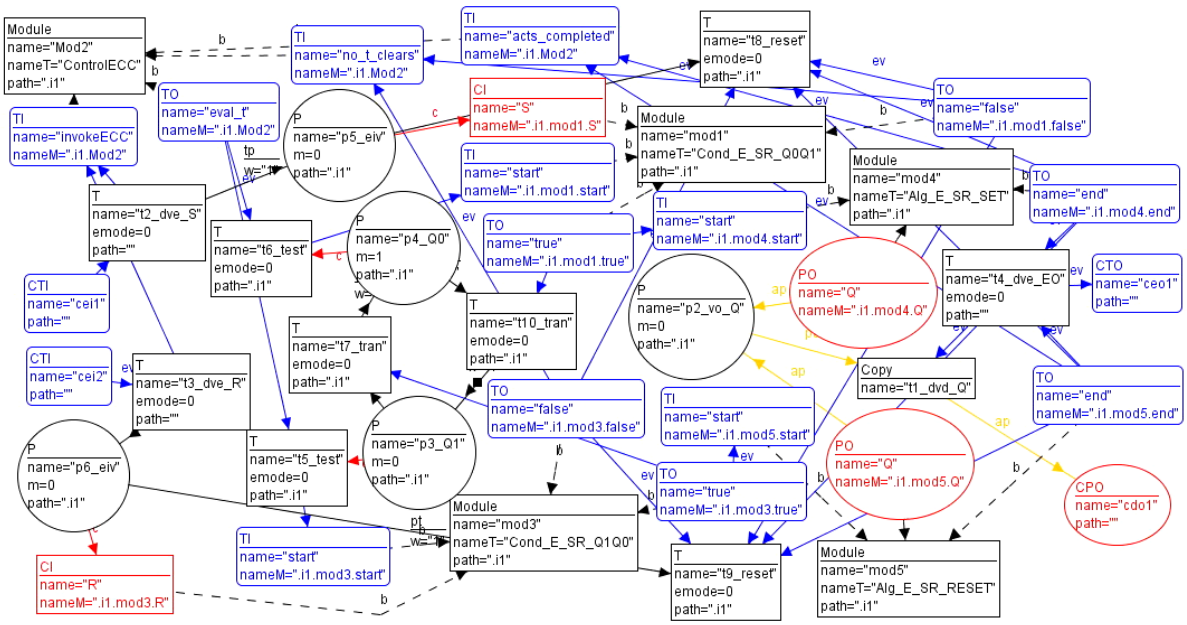


Рис. 5.49. Графовое представление модульной *aNCES*-сети, полученное в результате преобразований графовой модели системы ФБ, приведенной на рис. 5.47, 5.48, в системе *AGG*

Более традиционное представление данной *aNCES*-сети с использованием нотации сетей Петри представлено на рис. 5.50.

Следует отметить, что *NCES*-модуль типа *ConrtrolECC* на рис. 5.50 представляет управляющую часть интерпретатора *ECC*.

Модуль *CondEVENT* предназначен для вычисления условия, включающего только один событийный вход. Модули *Alg_E_SR_SET* и *Alg_E_SR_RESET* представляют алгоритмы установки выходной булевой переменной *Q* в значение «Истина» (1) и «Ложь» (0) соответственно.

Содержимое модуля *Alg_E_SR_SET* приведено на рис. 5.51. В данном случае булевы переменные представляются целочисленными значениями.

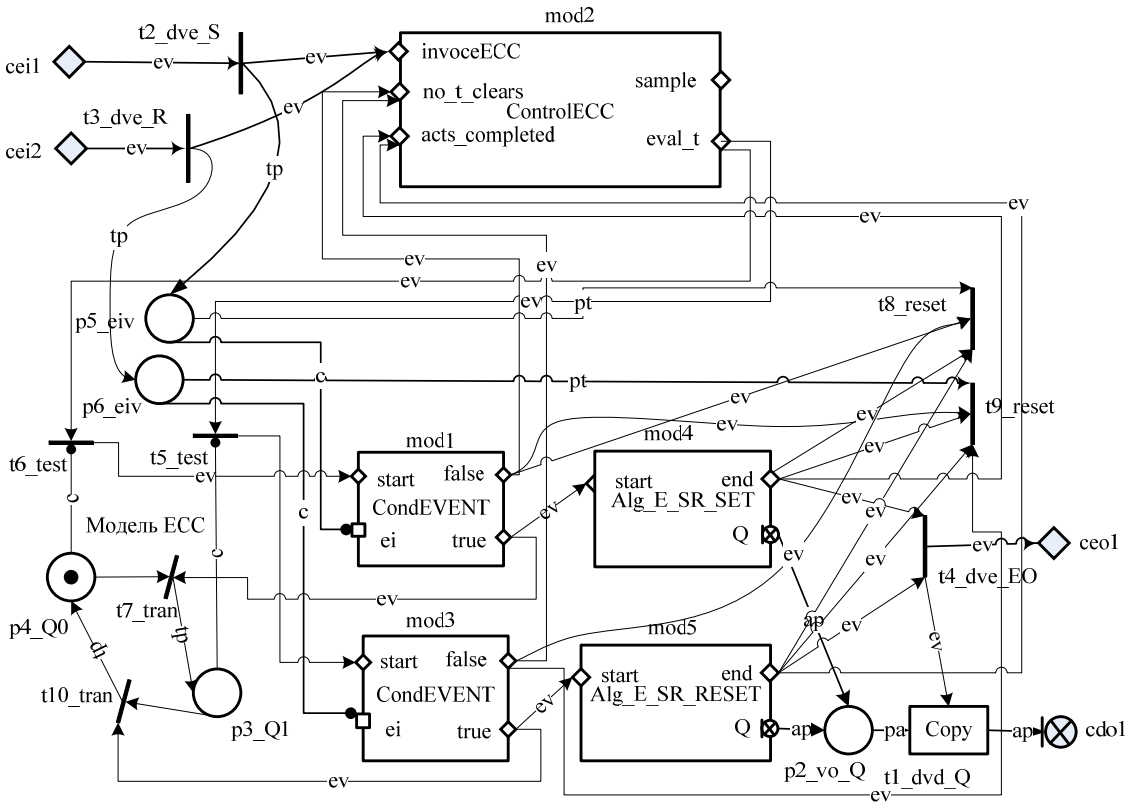


Рис. 5.50. Модульная *aNCES*-сеть,
представляющая систему ФБ на рис. 5.47, 5.48

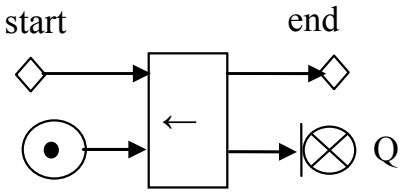


Рис. 5.51. Содержимое *aNCES*-модуля *Alg_E_SR_SET*

Одноуровневое представление сетевой модели может быть получено путем раскрытия модулей *aNCES*-сети. Данное преобразование в данном разделе не рассматривается. Более сложный пример трансформации системы ФБ можно найти в [20].

5.8. Реализация системы синтеза формальных моделей функциональных блоков

Структура программного обеспечения системы синтеза сетевых моделей ФБ представлена на рис. 5.52.

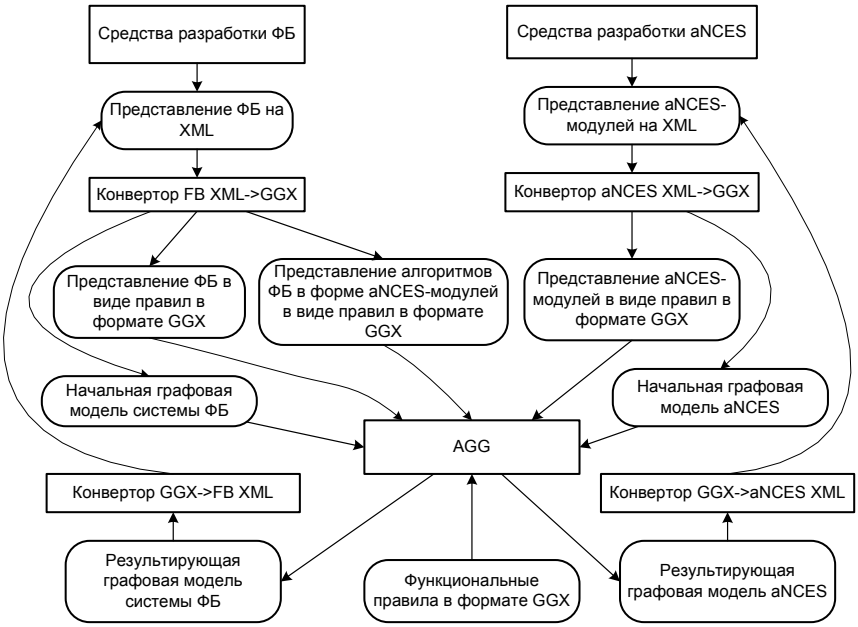


Рис. 5.52. Структура системы синтеза формальных моделей ФБ

Ядром системы является инструментальное средство *AGG*, поддерживающее трансформацию атрибутных графов, в совокупности с базой разработанных функциональных правил перезаписи графов. Основным форматом, используемым в *AGG*, является *GGX*-формат, основанный на *XML*. В файле *GGX* хранятся описания типов вершин и дуг, используемых при построении графов, описания самих графов и правил их перезаписи, а также опции системы.

Для связи *AGG* с системами проектирования на основе ФБ и *aNCES*, имеющими графические средства формирования, редактирования и отображения данных моделей, используются соответствующие *конверторы*. Как видно из рис. 5.52, связь между программными компонентами реализуется через *XML*-файлы. В данном случае программные компоненты являются независимыми. Возможен другой подход к построению системы, основанный на объединении всех программных компонентов в рамках некоторой оболочки, осуществляющей общее программное управление. Система *AGG* имеет программный интерфейс (*API*), ориентированный на *Java*, и таким образом, может легко интегрироваться в другие *Java*-приложения. К настоящему времени разработан действующий прототип системы синтеза сетевых моделей систем ФБ [57].

6. Рефакторинг диаграмм управления выполнением базисных функциональных блоков

В данном разделе рассматривается рефакторинг диаграмм *ECC* базисных ФБ как средство поддержки разработки промышленных управляющих приложений на основе стандарта IEC 61499. Основной целью рефакторинга является избавление диаграммы *ECC* от условных дуг без событий. Предлагается расширенный рефакторинг, позволяющий избавиться от потенциально-тупиковых состояний. Рефакторинг *ECC* реализован как набор правил графовых трансформаций. Прототип системы рефакторинга реализован в системе *AGG*. Рефакторинг *ECC* может помочь в автоматическом выполнении эквивалентных преобразований управляющих программ.

6.1. Модель диаграммы управления выполнением *ECC*

Диаграмма управления выполнением (далее – диаграмма *ECC* или просто *ECC*) определяет порядок выполнения операций в базисном ФБ и представляет собой диаграмму состояний особого вида. Для решения задач рефакторинга будем использовать упрощенную модель *ECC*, отличную от приведенной в разделе 3 и [120].

Определим диаграмму *ECC* как кортеж:

$$ECC = (S, R, E, C, A, D, f_E, f_C, f_A, f_P),$$

где $S = \{s_1, s_2, \ldots, s_n\}$ – множество вершин, представляющих *EC*-состояния; $R \subseteq S \times S$ – множество дуг, представляющих *EC*-переходы; $E = \{e_1, e_2, \ldots, e_m\}$ – множество событийных входов; $C = \{c_1, c_2, \ldots, c_k\}$ – множество сторожевых условий, определенных на множествах входных, выходных и внутренних переменных базисного ФБ; $A = \{a_1, a_2, \ldots, a_p\}$ – множество последовательностей *EC*-акций; $D \subseteq A \times C$ – отношение, определяющее зависимость условий *EC*-переходов от результатов выполнения *EC*-акций, $(a_i, c_j) \in D$, если выполнение a_i может изменить значение c_j. Следует заметить, что, как показывает практика, зависимость сторожевых условий от *EC*-акций случается весьма редко.

В соответствии с синтаксисом условий *EC*-переходов стандарта IEC 61499:

<Событийный вход> | <Сторожевое условие без событийного входа> | <Событийный вход> & <Сторожевое условие>

разобьем множество дуг R на следующие классы: R_E – событийных дуг, R_C – условных дуг и R_T – безусловных дуг, $R = R_E \cup R_C \cup R_T$; $R_E \cap R_C \cap R_T = \varnothing$. Событийная дуга ($E$-дуга) представляет EC-переход с событием; условная дуга (C-дуга) – EC-переход без события, имеющий сторожевое условие, отличное от тождественно истинного, а безусловная дуга (T-дуга) аналогична дуге второго типа, но сторожевое условие этой дуги тождественно истинно. В дальнейшем будем обозначать E- и T-дуги сплошной линией, а C-дуги – пунктирной. Над T-дугой будем ставить символ «t», над E-дугой – символ «e» (если это необходимо);

$f_E: R_E \to E$ – функция, назначающая E-дугам событийные входы;

$f_C: R_E \cup R_C \to C$ – функция, назначающая E- и C-дугам сторожевые условия;

$f_A: S \to A$ – функция, назначающая состояниям последовательности EC-акций.

$f_P: R \to \{1,2,...\}$ – функция, назначающая дугам нормализованные приоритеты, причем $f_P = \bigcup_{s \in S} f_P^s$, где $f_P^s: R^s \to \{1,2,\ldots,|R^s|\}$ – функция (биекция) назначения приоритетов, относящаяся к вершине s, R^s – это множество всех дуг, которые исходят из вершины s. Приоритет дуги $r1$ выше приоритета дуги $r2$, если $f_p(r1) < f_p(r2)$.

В диаграмме ECC стандарта IEC 61499 приоритет EC-перехода явно не указывается. Для его задания используется позиционный принцип описания: чем выше в XML-документе стоит описание EC-перехода, тем он приоритетнее по отношению к другим EC-переходам, описание которых стоит ниже по тексту.

6.2. Модели выполнения *ECC*

В соответствии со стандартом IEC 61499 управление выполнением ECC осуществляет специальный интерпретатор (так называемая OSM-машина), диаграмма состояний которого представлена на рис. 1.5. Как было отмечено в [219, 247] определение интерпретации ECC в стандарте является неполным и, следовательно, двусмысленным. Оно, например, допускает два различных подхода к оценке EC-переходов без событий. В соответствии с первым подходом, EC-переход без событий разрешен, если: 1) он связан с обработкой какого-то конкретного сигнала, инициировавшего текущий «прогон» (*Single Run*) ФБ; 2) EC-переход выходит из текущего EC-состояния; 3) сторожевое условие этого перехода истинно. Второй под-

ход не связывает *EC*-переход с каким-либо конкретным событием. В этом случае разрешенность *EC*-перехода связана с истинностью соответствующего сторожевого условия. Назовем сторожевое условие перехода без события в первом случае *пассивным*, а во втором случае – *активным*. В литературе отражены оба подхода. Первый подход представлен в [219]. Второй подход представлен в работе, вводящей модель последовательного выполнения ФБ [247]. В дальнейшем будем рассматривать только первую модель выполнения *ECC*, в которой существует прямая необходимость в рефакторинге диаграмм *ECC*.

Определение 6.1. Потенциально-тупиковым (по условиям) состоянием (ПТ-состоянием) в модели выполнения *ECC* с пассивными сторожевыми условиями назовем *EC*-состояние s_i такое, что $\forall j \in 1 \ldots n$ $[(s_i, s_j) \in R \rightarrow (s_i, s_j) \in R_C]$. Иными словами, s_i является ПТ-состоянием, если все выходящие из него дуги являются условными. Если интерпретатор *ECC* совершил переход $t2$ (см. рис. 1.5) в то время, как диаграмма *ECC* находилась в ПТ-состоянии, то это состояние становится тупиковым. В дальнейшем никакие входные сигналы ФБ не смогут его изменить.

Определение 6.2. Две *ECC* называются *функционально эквивалентными* (в рамках определенной модели выполнения *ECC*), если при любых последовательностях входных событий и сопутствующих им наборов значений входных переменных, обе *ECC* выполняют одни и те же последовательности *EC*-акций.

6.3. Общий подход к рефакторингу и исправлению диаграмм *ECC*

Рефакторинг диаграмм *ECC* используется для избавления от *C*-дуг полностью, если это возможно, или, по крайней мере, для минимизации числа *C*-дуг и удаления ПТ-состояний, появляющихся в результате этой минимизации. В соответствии с этим будем различать рефакторинги первого и второго типов (рефакторинги 1 и 2). *Рефакторинг 1* помогает разработчику получить несколько иную точку зрения на разработанную *ECC*, что в ряде случаев (на основе визуального анализа) может помочь ему переосмыслить и перепроектировать диаграмму *ECC*. *Рефакторинг 2* идет дальше и усовершенствует *ECC* путем удаления ПТ-состояний. Таким образом, результаты рефакторинга 2 базируются на результатах рефакторинга 1. Прямую практическую значимость имеет рефакторинг 2. Сле-

дует отметить, что рефакторинг 2 не обеспечивает эквивалентность исходной и результирующей диаграмм *ECC*.

Назовем *CT*-сетью диаграммы *ECC* подграф, содержащий дуги только из $R_C \cup R_T$; но не из R_E. В общем случае данный граф будет несвязным. Соответственно, *T*-сетью *ECC* будем называть подграф, содержащий дуги из R_T. Предполагается, что исходная *CT*-сеть ациклична. Наличие циклов свидетельствует о некорректности *ECC*. Хотя в общецелевом программировании циклические структуры нашли широкое распространение, в диаграммах *ECC* функциональных блоков итерационные процедуры рекомендуется реализовывать в алгоритмах, а не в *ECC*.

Введем $ES = \{(s, s') \in R_E \mid \exists(s', s'') \in R_C \cup R_T\}$ – множество *E*-дуг, образующих путь длиной 2 с одной из *C*- или *T*-дуг *CT*-сети. Очевидно, что данные *E*-дуги имеют *C*- или *T*-дуги в качестве последователей. Назовем эти *E*-дуги источниками. Они будут основным исходным пунктом действий по рефакторингу.

Ниже представлена общая идея избавления *ECC* от событийных дуг. Она основывается на понятии достижимости последовательностей *EC*-акций при интерпретации *ECC*. Пусть $(s_0, s_1) \in EC$ – событийная дуга, за которой следует путь $s_1, s_2, \ldots, s_k$ в *CT*-сети. Каждому *EC*-состоянию s_i ($i = 1, \ldots, k$) соответствует последовательность *EC*-акций a_i. В примере на рис. 6.1 путь составлен только из *C*-дуг, что не меняет сути дела.

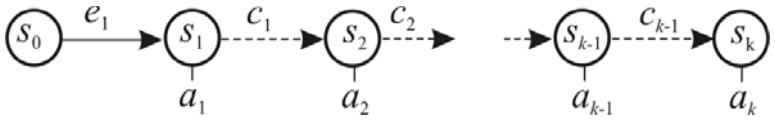


Рис. 6.1. Путь, состоящий из дуги-источника и последующих *C*-дуг

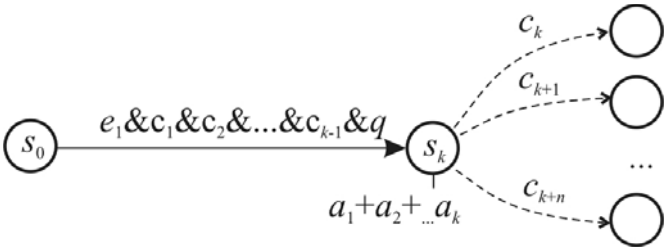


Рис. 6.2. *E*-дуга, представляющая путь из рис. 6.1

В случае, когда условия *EC*-переходов являются независимыми от предшествующих им *EC*-акций при выполнении ФБ, этот путь

можно заменить одной *E*-дугой (s_0, s_k), сторожевое условие которой составляется из сторожевых условий дуг c_i, (i = 1, …, k – 1), входящих в путь и так называемого *условия сохранения целевого состояния q* (рис. 6.2). Последовательность *EC*-акций, выполняемых в целевой вершине s_k, определяется как конкатенация (сцепление) последовательностей *EC*-акций вершин, входящих в путь.

Условие q сохранения некоторого состояния s_k определяется как конъюнкция отрицаний сторожевых условий исходящих *C*-дуг: $\&_{(s_k,s_j)\in R_C} \overline{f_C(s_k,s_j)}$. Например, для состояния s_k на рис. 6.2 условие сохранения состояния равно $\overline{\tilde{n}_k} \,\&\, \overline{c_{k+1}} \,\&\, ... \,\&\, \overline{c_{k+n}}$. Если из состояния s_k исходит *T*-дуга, то это состояние является транзитным и переход в него невозможен, поскольку интерпретатор *ECC* все время будет «проскакивать» через это состояние.

Если в целевое состояние s_k пути входят какие-то другие дуги, то путь s_1, s_2, …, s_k, начинающийся из дуги (s_0, s_1), следует заменять двумя дугами (s_0, s_{k-1}) и (s_{k-1}, s_k), первая из которых в целом идентична *E*-дуге из рис. 6.2, а вторая дуга является *T*-дугой. Необходимость второй дуги определяется тем, что входящие в целевое состояние s_k дуги определяют альтернативные пути, которым соответствует суммарная последовательность *EC*-акций, в общем случае отличная от последовательности *EC*-акций, определяемых рассматриваемым путем. Поэтому приписывание целевому состоянию s_k последовательности *EC*-акций какого-то одного из путей было бы неверным. Выход из этого положения – приписать все *EC*-акции пути, за исключением *EC*-акций целевой вершины, промежуточной вершине s_{k-1} и соединить ее безусловной дугой с целевой вершиной (рис. 6.3). Назовем вершину s_{k-1} *представителем* вершины s_k, поскольку большинство *EC*-акций, которые выполнялись бы в состоянии s_k, выполняются именно в этой промежуточной вершине.

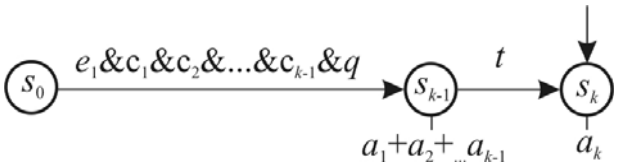


Рис. 6.3. *E*- и *T*-дуги, представляющие путь из рис. 6.2

Для заданной дуги r = (s_i, s_j)$\in ES$ введем операцию *связывания с произвольной вершиной s_k из CT-сети*. Данная операция заключается в нахождении всех путей из вершины s_j в вершину s_k в *CT*-сети и за-

меной их *E*-дугами (или парами дуг типа (*E*,*T*)), как описано выше и проиллюстрировано на рис. 6.3. В общем случае, результатом такой операции является *гамак* (рис. 6.4). Все *E*-дуги, исходящие из вершины s_i, имеют одинаковые имена событийных входов $f_E(s_i, s_j)$ (на рисунке $f_E(s_i, s_j) = e_m$). Следует, однако, отметить, что операция связывания дуги с вершиной *CT*-сети не всегда возможна.

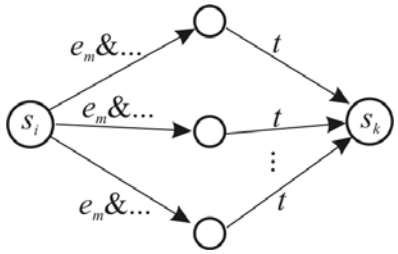


Рис. 6.4. Результат связывания дуги-источника и вершины *CT*-сети

Назовем *связыванием дуги* $r_i \in ES$ *с CT-сетью* операцию связывания этой дуги со всеми вершинами *CT*-сети. Для полного избавления *ECC* от *C*-дуг необходимо связать все дуги из множества *ES* с соответствующей *CT*-сетью и далее все *C*-дуги удалить. Можно утверждать, что любую ациклическую *CT*-сеть без зависимостей между *EC*-акциями и сторожевыми условиями (т.е. при $D = \varnothing$) можно преобразовать к виду, свободному от *C*-дуг. Полученную в результате таких преобразований *T*-сеть в совокупности с соответствующими *E*-дугами можно назвать графом достижимости наборов *EC*-акций в исходной *CT*-сети. Очевидно, что преобразованная в соответствии с предложенным методом *ECC* будет эквивалентна исходной.

При проведении рефакторинга важно не только получить новую структуру *ECC*, вычислить сторожевые условия и наборы выполняемых *EC*-акций, но и определить приоритеты дуг в обновленной *ECC*. Метод определения приоритетов дуг рассмотрим на примере (рис. 6.5 и 6.6). На рис. 6.5 приведен пример *ECC* в виде бинарного дерева с одной *E*-дугой в качестве «движущей силы». Будем идентифицировать *C*-дуги на данном рисунке соответствующими им сторожевыми условиями. В исходной *ECC* используются нормализованные приоритеты. Они написаны рядом с соответствующими дугами. Очевидно, что самый высокий приоритет имеет путь c_1, c_3. При истинности всех условий $c_1 ... c_6$ будет реализовываться именно он. Путь c_2, c_6 – наименее приоритетный. Он будет реализовываться только при истинности условий c_2 и c_6 и ложности всех остальных условий. Из приведенного примера можно вывести

правило, согласно которому чем ближе к началу пути находится дуга, тем большее влияние оказывает она на «суммарный» приоритет пути. Исходя из этого предлагается использовать составные приоритеты (в виде кортежа) с заданным на них лексикографическим порядком $\prec$. Составной приоритет формируется как конкатенация приоритетов дуг пути от начальной вершины до целевой. Для представления составных приоритетов может использоваться точечная нотация. Преимуществом составных приоритетов является легкость их вычислений.

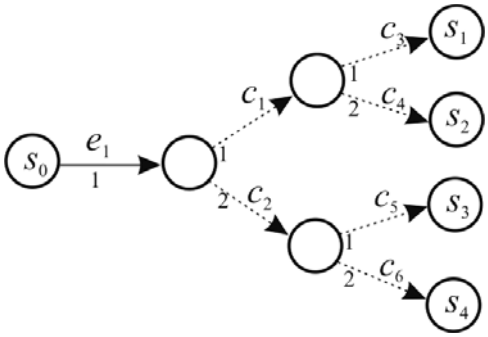


Рис. 6.5. Диаграмма *ECC* в виде бинарного дерева

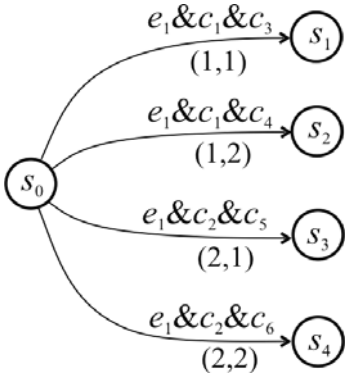


Рис. 6.6. Преобразованная диаграмма *ECC* (из рис. 6.5) с составными приоритетами дуг

На рис. 6.6 представлена преобразованная *ECC*. Вычисленные составные приоритеты записаны под соответствующими дугами. Как видно из рис. 6.6, дуга (s_0, s_1) самая приоритетная, поскольку $\forall pr \in \{(1,2), (1,3), (2,1), (2,2)\}[(1,1) \prec pr]$.

Возможен вариант рефакторинга, при котором приоритеты дуг становятся излишними. Это достигается тем, что дугам приписываются такие сторожевые условия, которые разрешают только одну дугу, выходящую из какой-либо вершины. При истинности сторожевого условия такой дуги сторожевые условия других дуг, выходящих из той же вершины, становятся ложными. Результат преобразования диаграммы *ECC*, представленной на рис. 6.5, в соответствии с данным рефакторингом приведен на рис. 6.7. Недостатком метода является громоздкость сторожевых условий, что может привести к существенному увеличению времени их вычисления.

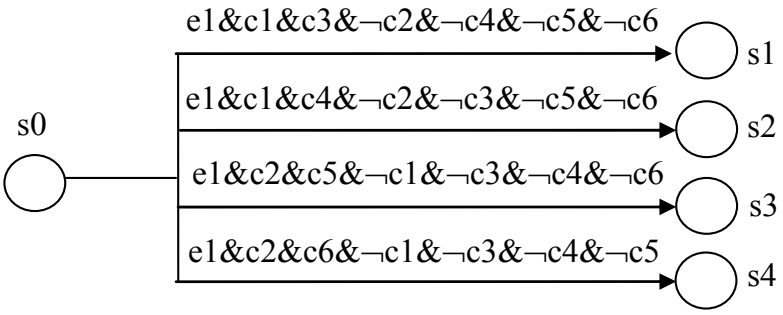


Рис. 6.7. Преобразованная диаграмма *ECC* (из рис. 6.5) без приоритетов дуг

Следует также отметить, что существует два подтипа рефакторинга в зависимости от того, из каких *EC*-состояний производится «трассировка сигнала». В первом случае активной дугой является *E*-дуга, выходящая из любого *EC*-состояния, во втором случае – только та *E*-дуга, которая выходит из *EC*-состояния, достижимого из начального. Второй вариант назовем рефакторингом с учетом достижимости из начального состояния. В дальнейшем рассматривается первый вид рефакторинга, являющийся более общим.

6.4. Рефакторинг на основе графотрансформационного подхода

Ниже предлагается подход к решению задачи рефакторинга на основе локальных эквивалентных преобразований *ECC* с использованием системы перезаписи типизированных атрибутных графов (ТАГ). Одно эквивалентное преобразование может выражаться в общем случае применением не одного, а нескольких правил. Процесс локальных эквивалентных преобразований графовой модели *ECC* проходит в рамках вычислений, описанных в предыдущем подразделе.

Основные правила трансформации связаны с обработкой пары смежных дуг (s_i, s_j) и (s_j, s_k) и формированием на их основе новой прямой дуги, ведущей в состояние s_k или в его представителя. Смыслом большинства правил является построение множества достижимых (некоторым сигналом) наборов *EC*-акций путями длиной 2. Дуги, входящие и выходящие из вершин s_i, s_j и s_k, представляют контекст применения правила.

Все правила можно разделить на следующие группы: 1) правила предварительной обработки (корректировки); 2) правила нара-

щивания графа и 3) правила очистки графа. Упрощенный алгоритм трансформации *ECC*, основанной на правилах, включает следующие шаги: 1) применяются правила первой группы до тех пор, пока это возможно. Для этого делается сопоставление исходной *ECC* с левой частью каждого правила. Если такое сопоставление найдено, соответствующий подграф *ECC* трансформируется в подграф правой части данного правила; 2) применяются правила второй группы аналогичным способом; 3) применяются правила третьей группы.

Правила первой группы приводят исходную *ECC* к некоторой нормализованной форме. Примеры правил этого типа представлены на рис. 6.8 – 6.10. Правило слияния параллельных *C*-дуг исключает наличие нескольких однонаправленных *C*-дуг между парами вершин (рис. 6.8). Правило удаления мертвых *E*-дуг удаляет дуги, выходящие из состояния, из которого также выходит хотя бы одна *T*-дуга (рис. 6.9). Переход по данным *E*-дугам никогда не осуществится, поскольку интерпретатор *ECC* немедленно переведет автомат из текущего состояния в целевое состояние *T*-дуги.

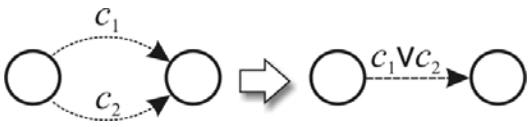


Рис. 6.8. Правило слияния параллельных *C*-дуг

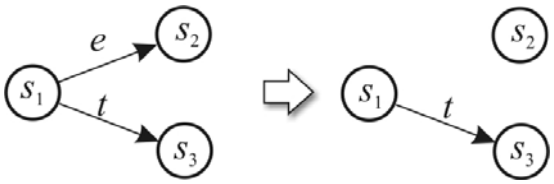


Рис. 6.9. Правило удаления мертвых *E*-дуг

Правило на рис. 6.10 удаляет *T*-дугу, которая является мертвой вследствие того, что она менее приоритетна по отношению к другой *T*-дуге, выходящей из той же вершины. На рис. 6.10 под *pr* понимается приоритет дуги. Над стрелкой между левыми и правыми частями правил надписано условие применения правила: $x < y$.

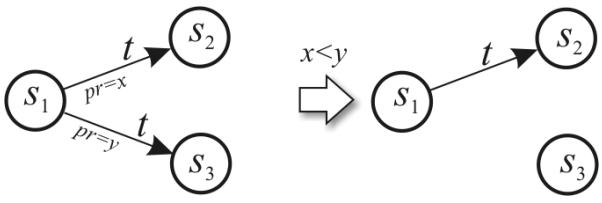


Рис. 6.10. Правило удаления мертвой неприоритетной *T*-дуги

Правила наращивания графа (правила второй группы) осуществляют основные преобразования и по выполняемой функции делятся на несколько групп. Внутри группы правила различаются только

контекстом. На рис. 6.11–6.14 схематично представлены основные группы правил данного вида в виде некоторых шаблонов. Контекстные состояния обозначены маленькими кружками. Тип контекстных дуг не уточняется. Крест на контекстной дуге в левой части правила означает, что дуга запрещена. Таким путем кратко представляются *NAC*-условия. Черточка на дуге в правой части правила означает, что дуга является использованной и в последующем будет удалена. Для *E*-дуги вычисляется два условия: 1) условие достижения целевого состояния (включающее также имя событийного входа) и 2) условие распространения сигнала за целевое состояние. Первое условие пишется над дугой, а второе – под дугой. Полное текущее условие для *E*-дуги определяется как конъюнкция первого условия и отрицания второго условия.

Цель правила *R*1 (рис. 6.11) – удаление условной дуги (s_2, s_3), которая следует за событийной дугой. Это достигается путем добавления прямой дуги (s_1, s_3), модификации условия под дугой (s_1, s_2) и передачи *EC*-акций от s_2 к s_3. Это правило может быть применено к той части *ECC*, где вершины s_2 и s_3 не имеют входящих дуг (что указывается контекстными дугами с крестиками в левой части правила).

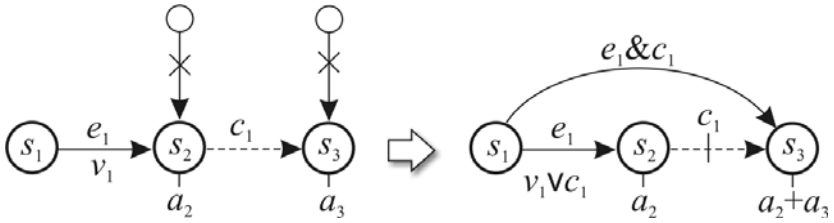


Рис. 6.11. Правило распространения сигнала
(на линейном участке)(*R*1)

Правило *R*2 (рис. 6.12) имеет сходную цель, но может быть применено к тем частям *ECC*, в которых s_3 имеет входящую дугу. Чтобы избежать конфликта, который может потенциально возникнуть, когда *EC*-акции переносятся в эту вершину из других частей, вводится промежуточное состояние s_4, которому назначаются *EC*-акции состояния s_2.

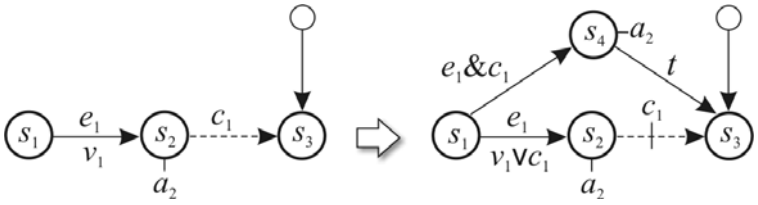


Рис. 6.12. Правило вхождения в соединитель (*R*2)

Цель правила $R3$ (рис. 6.13) – получить «чистый» путь между двумя вершинами (s_1 и s_3), без захода в него других дуг. Если бы в вершину пути заходила посторонняя дуга, то был бы возможен конфликт при переносе акций в промежуточную вершину (s_2) из разных путей.

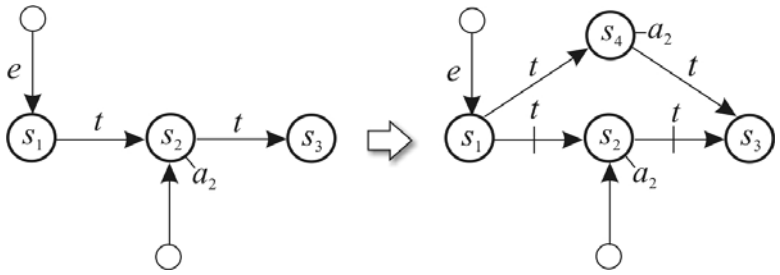


Рис. 6.13. Правило обхода соединителя ($R3$)

Как правило, вновь создаваемая (дочерняя) дуга имеет двух родителей – пары следующих друг за другом смежных дуг. При этом приоритет дочерней дуги полагается равным конкатенации приоритетов родительских дуг, причем первым идет приоритет дуги, стоящей в этой паре первым.

В правиле избавления от обратной C-дуги ($R4$)(рис. 6.14) в правой части правила в отличие от большинства правил имеются две дочерние дуги (s_1, s_3) и (s_1, s_2), они имеют одинаковый приоритет, равный приоритету x родительской дуги. Это не имеет значения, так как они являются взаимоисключающими по условию c_1.

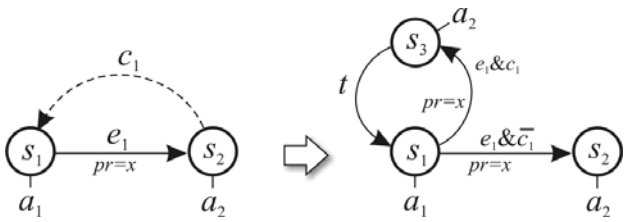


Рис. 6.14. Правило избавления от обратной C-дуги ($R4$)

Условием применения правил $R1$, $R2$ и $R4$ является отсутствие зависимостей между EC-акциями и условиями EC-переходов, т.е. $(a_2, c_1) \notin D$.

В отличие от правил первой и второй групп, правила очистки графа являются специфическими для каждого из типов проводимого рефакторинга. В качестве примера правил данной группы можно привести правила удаления E-дуги и правило удаления изолирован-

ной вершины (рис. 6.15). Согласно первому из правил из обрабатываемого графа удаляются те *E*-дуги, которые использованы для генерации других дуг и не входят непосредственно в другие *E*-дуги и не являются висячими и не имеют обратной *T*-дуги.

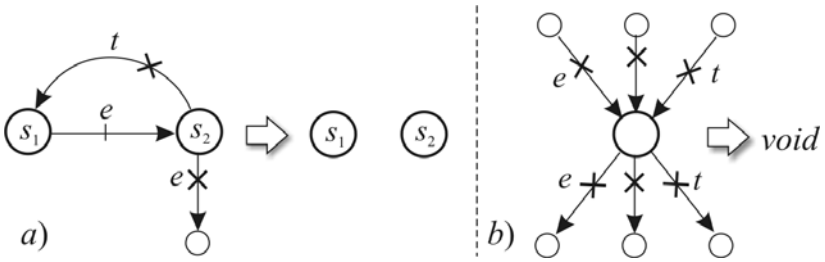


Рис. 6.15. Правила удаления:
а – *E*-дуги (*R*5); *б* – изолированной вершины (*R*6)

6.5. Примеры рефакторинга *ECC*

Для иллюстрации предложенного метода рефакторинга ниже рассматривается процесс рефакторинга нескольких диаграмм *ECC*. Сначала рассмотрим простую диаграмму *ECC* и ее преобразование (рис. 6.16).

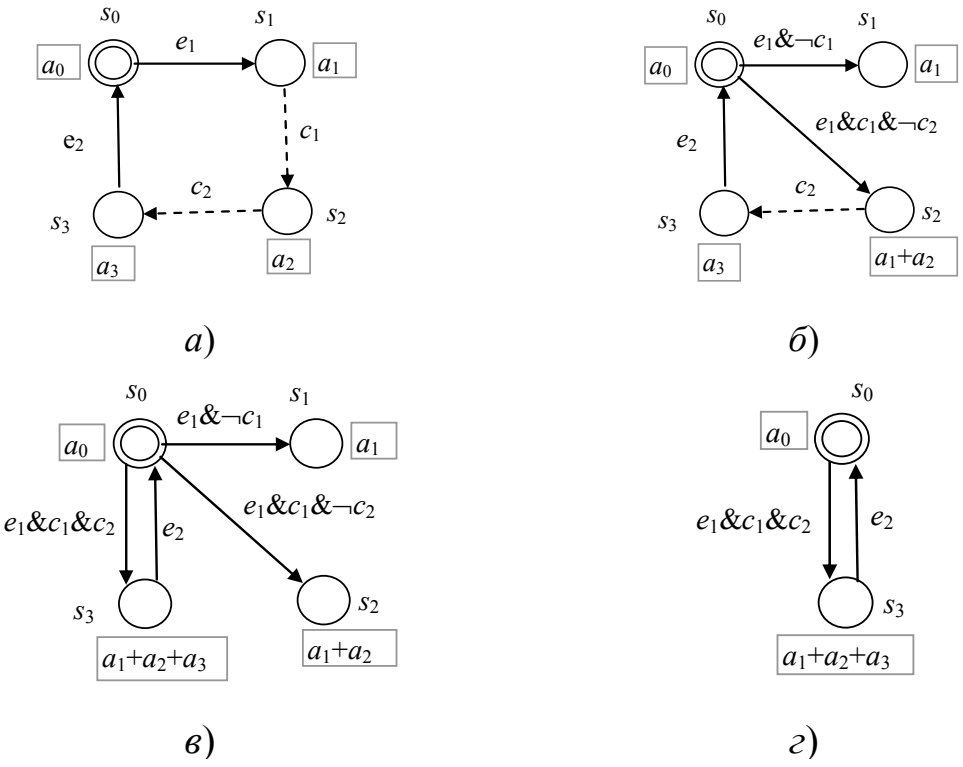


Рис. 6.16. Простой пример рефакторинга *ECC*:
а – исходная *ECC*; *б* – промежуточная *ECC*; *в* – результат рефакторинга 1;
г – результат рефакторинга 2

Процесс рефакторинга 1 определяется цепочкой правил $R1$, $R1$. Упрощения (для облегчения понимания сути) заключаются в следующем: 1) считаем, что использованные C-дуги уничтожаются непосредственно в правиле $R1$ (а не в других правилах); 2) над E-дугой пишется одно полное условие (а не два условия, как в правилах). После первого применения правила $R1$ получается промежуточная *ECC*, представленная на рис. 6.16,*б*. После второго применения правила $R1$ получается результирующая для рефакторинга 1 *ECC* (рис. 6.16,*в*). Она эквивалентна исходной *ECC*. В результирующей *ECC* ПТ-состояния представлены в виде тупиковых вершин s_1 и s_2.

Рефакторинг 2 совершается путем дополнительного применения правил $R5$ и $R6$. Результирующая *ECC* для этого случая представлена на рис. 6.16,*г*. Эта *ECC* не эквивалентна исходной *ECC*, поскольку в ней отсутствуют ПТ-состояния s_1 и s_2. Это приводит к тому, что, например, при входном сигнале e_1, истинности условия c_1 и ложности условия c_2 (кратко $e_1 \& c_1 \& \neg c_2$) в результирующей *ECC* не выполняется ни одна *EC*-акция, в то время как в исходной *ECC* выполняются последовательности *EC*-акций a_1 и a_2, после чего работа *ECC* «замораживается». Несмотря на то, что исходная и результирующая диаграммы *ECC* не эквивалентны, результирующая диаграмма *ECC* на рис. 6.16,*г* с большой долей уверенности соответствует замыслам разработчика. В приведенном случае разработчик, по всей видимости, предполагает, что ожидается приход события e, после чего будет выполнена *EC*-акция a_1, затем ожидается истинность условия c_1, после чего выполнится *EC*-акция a_2 и т.д. Результаты рефакторинга 1 показывают, что спроектированная *ECC* не вполне корректна. «Неожиданные» тупиковые состояния могли возникнуть из-за неточного понимания семантики *ECC* (с пассивными сторожевыми условиями), которая отличается от семантики классической автоматной модели. Следует отметить, что в примере на рис. 6.16 приоритеты дуг не имеют никакого значения вследствие взаимоисключающих сторожевых условий.

Более сложный пример диаграммы *ECC* и результат ее рефакторинга второго типа представлен на рис. 6.17. Нормализованные приоритеты дуг в виде чисел поставлены рядом с дугами. Для краткости для результирующей *ECC* опущены приоритеты тех дуг, которые являются единственными, выходящими из вершины.

Третий пример взят из практики [246]. В этом случае объектом исследования является пневматический цилиндр с несколькими

управляющими кнопками и световым устройством безопасности (рис. 6.18,*а*). Контроллер этой системы реализован в виде ФБ cylinder. Его интерфейс показан на рис. 6.18,*б*, а управляющая логика реализована в виде *ECC* на рис. 6.18,*в*. Для разработки и реализации системы управления использовался инструментарий фирмы *nxtControl* [187].

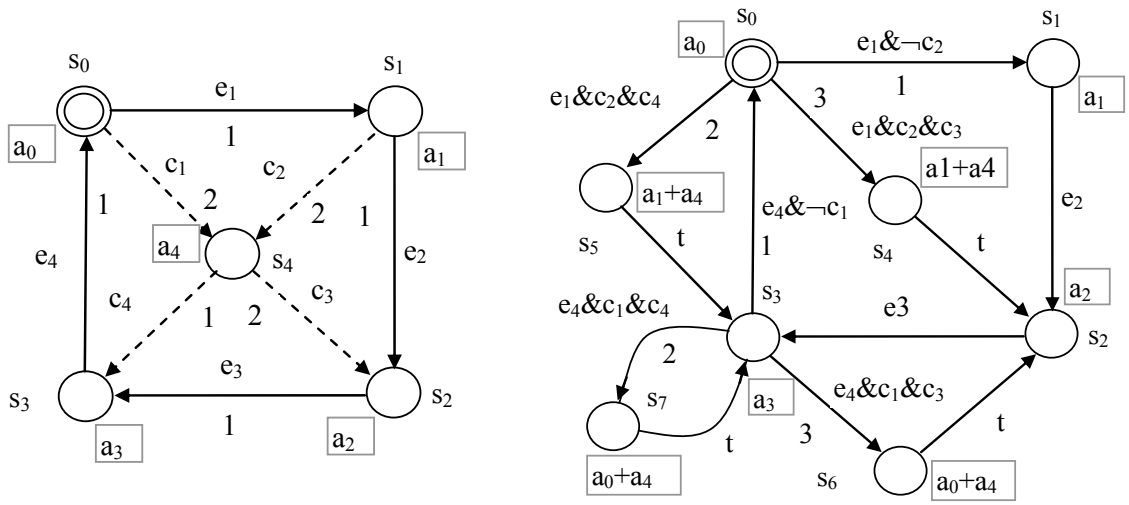


Рис. 6.17. Диаграмма *ECC* (слева)
и результат ее рефакторинга 2 (справа)

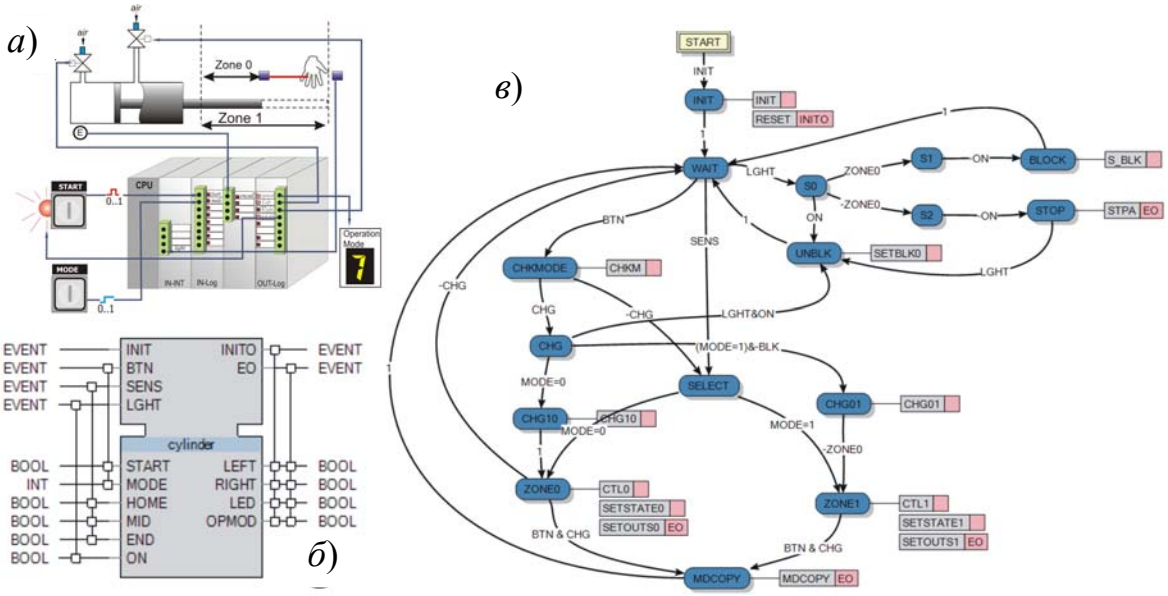


Рис. 6.18. Система управления пневматическим цилиндром:
а – мнемосхема; *б* – интерфейс функционального блока контроллера;
в – диаграмма *ECC* функционального блока контроллера

Система функционирует следующим образом. Цилиндр движется (назад и вперед) или из левой позиции в среднюю позицию,

или из левой позиции в конечную позицию в зависимости от выбранного режима. Режим выбирается путем нажатия кнопки MODE, которая имеет два положения, одно из которых соответствует значению 0, а другое – 1. При пересечении луча света некоторым объектом операция должна приостановиться до тех пор, пока объект не покинет зону безопасности.

«Световой» сигнал связан с особым входным портом управляющего устройства, которое генерирует прерывание при каждом изменении значения состояния детектируемого луча. В терминах ФБ сигнал прерывания соответствует событийному входу LIGHT блока cylinder.

ФБ cylinder имеет шесть информационных логических входов, соответствующих кнопкам START и MODE, трем дискретным позициям цилиндра (HOME, MID, END) и логическому состоянию шторки (сенсора) для луча. Кроме того, существуют четыре событийных входа. Событийный вход INIT используется для инициализации. Сигнал на вход BTN посылается при изменении состояния кнопки (нажата/отжата). Сигнал на входе SENS появляется, когда цилиндр прибывает на одну из трех дискретных позиций. Сигнал на входе LIGHT является индикатором того, что изменился статус сенсора луча. ФБ cylinder выдает данные через следующие информационные выходы: LEFT и RIGHT – о положении исполнительного механизма; LED – для подсветки кнопки START в те моменты, когда необходима «чувствительность» этой кнопки к нажатию; OPMODE – для отображения текущего режима (т.е. зоны 0 или 1, см. рис. 6.18).

Диаграмма *ECC* контроллера (см. рис. 6.18,*в*) определяет как последовательную логику, реализующую управление движением цилиндра, так и реакцию на прерывания. Существенная часть управляющей логики инкапсулирована в алгоритмы, выполняемые в *EC*-состояниях. Например, алгоритмы CTL0 и CTL1 (состояния ZONE0 и ZONE1 соответственно) написаны на языке диаграмм Ладдера (языке LD) [162]. Для краткости их код не представлен, но их основная функция – вычисление значений логических сигналов для исполнительных механизмов.

В диаграмме *ECC*, представленной на рис. 6.18,*в*, существуют тупиковые состояния. Например, после прерывания, исходящего от сенсора луча (появившегося в результате вторжения объекта) в то время, когда поршень был в зоне 0, *ECC* проходит последовательность состояний *WAIT*→*S*0→*S*1→*BLOCK*→*WAIT*, устанавливая внутреннюю переменную *BLK* в значение *TRUE* (предполагается,

что это значение проверяется, если оператор изменяет режим *MODE* в 1). Однако после того, как вторгнувшийся объект был удален, эта *ECC* проходит последовательность *WAIT*→*S*0→*S*1 и останавливается в состоянии *S*1 навсегда, даже если и будет истинным условие *ON* = 1. Это может произойти, поскольку переход *S*0→*S*1 имеет больший приоритет по сравнению с переходом *S*0→*UNBLK*.

Для избавления от тупиковых ситуаций достаточно применить к диаграмме *ECC*, приведенной на рис. 6.18,*в*, правило *R*1 для слияния *E*- и *C*-дуги (рис. 6.19).

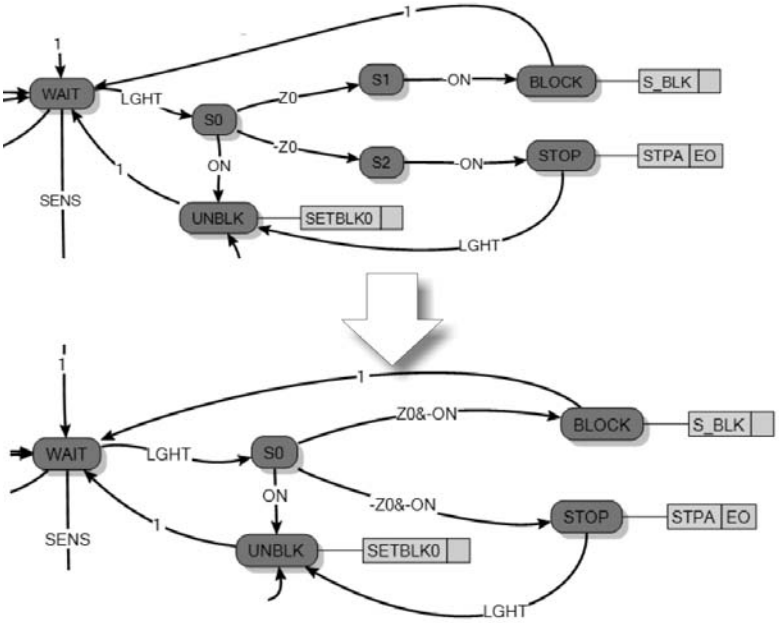


Рис. 6.19. Применение правила *R*1 для избавления
от потенциально-тупиковых состояний *S*1 и *S*2 в ФБ cylinder

6.6. Реализация системы рефакторинга в *AGG*

Прототип системы рефакторинга *ECC* был разработан на основе системы трансформации графов *AGG* с применением языка *Java*. Система *AGG* представляет основанную на правилах среду визуального программирования, использующую алгебраический подход к трансформации графов [75, 222]. Правила перезаписи графов могут содержать *NAC*- и контекстные условия. Полезным качеством *AGG*, отсутствующим в других системах, является то, что правила могут быть проверены, используя анализ критических пар и проверку консистентности. *AGG* поддерживает спецификацию типизированного графа с кардинальностью и атрибутами. Типы атрибутов заимствованы из языка *Java*.

Метамодель диаграммы *ECC*, используемая в прототипе системы рефакторинга, представлена в виде ТАГ, сформированного в *AGG* (рис. 6.20). Как видно из данного рисунка, метамодель *ECC* состоит, по сути, из единственной вершины типа *S* (представляющей *EC*-состояние) и четырех петель, представляющих *E*-, *C*-, *T*-, а также *p*-дуги. Дуги типа *p* являются вспомогательными и используются обычно для связывания родительского узла и вновь созданного дочернего узла. Метамодель *ECC* в форме ТАГ используется в *AGG* как средство контроля графовых преобразований.

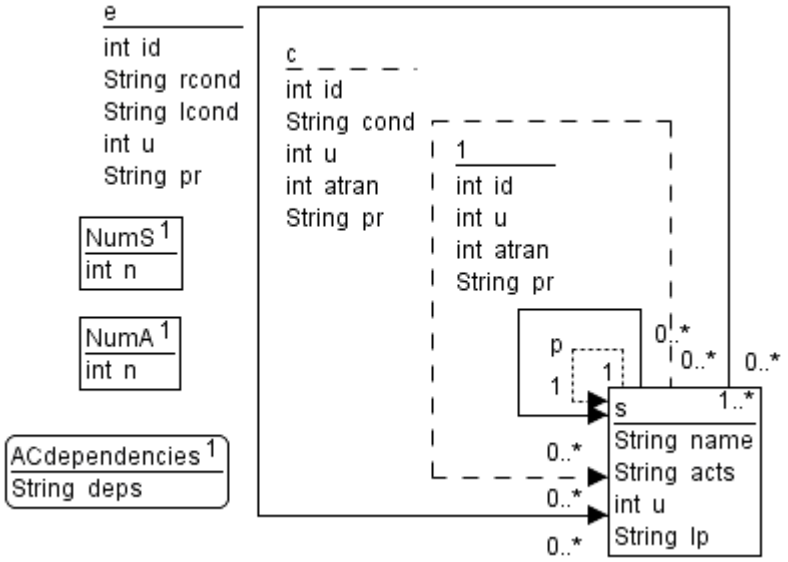


Рис. 6.20. Метамодель *ECC* в виде ТАГ

EC-состояние имеет следующие атрибуты: *name* – имя *EC*-состояния, *acts* – набор прикрепленных *EC*-акций, *lp* – список пар дуг, участвующих в генерации дуг через данную вершину. Часть атрибутов дуг общая, а часть – специфическая. Все дуги имеют следующие атрибуты: *id* – уникальный идентификатор, *u* – признак использования (*u* = 1 – дуга использована, *u* = 0 – не использована), *pr* – приоритет. *C*- и *T*-дуги имеют атрибут *atran* – признак передачи акций по дуге (*atran* = 1 – передача *EC*-акций производилась, *atran* = 0 – не производилась). Атрибут *cond C*-дуги определяет сторожевое условие. Для *E*-дуги формируются два условия: условие достижения состояния (атрибут *rcond*) и условие ухода из состояния (атрибут *lcond*). Как правило, сторожевое условие *E*-дуги вычисляется как конъюнкция условия достижения состояния и отрицания условия выхода из состояния. Для упрощения реализации механизма составных приоритетов полагается, что они символьного типа. Для получения составного приоритета используется операция конкатенации

строк. Отношение лексикографического порядка реализуется операцией сравнения строк языка *Java*.

В системе используется три типа вспомогательных вершин. Вершина типа *NumS* используется для нумерации вновь создаваемых вершин, вершина типа *NumA* – для нумерации вновь создаваемых дуг, а вершина *ACdependencies* содержит предопределенную «базу данных» существующих зависимостей «*EC*-акции – Условия *EC*-переходов».

Система рефакторинга содержит около 35 правил. На рис. 6.21 в качестве примера приведено одно правило системы *AGG*, шаблон которого представлен на рис. 6.11.

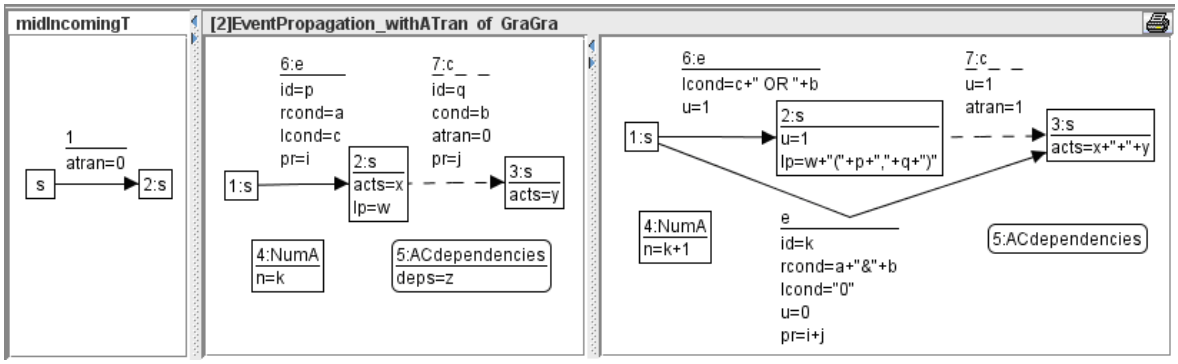


Рис. 6.21. Правило распространения сигнала (*R*1, рис. 6.11), представленное в *AGG*

Левая часть правила изображена в центре, правая часть – в правой части окна. Одно из условий *NAC* приведено в левой части окна. Кроме приведенного *NAC*, имеются еще четыре *NAC*-условия. Данное правило имеет следующие условия применения, выраженные с использованием языке *Java*: 1) w.*indexOf*(“(“+*p*+”,”+*q*+”)”)<0; 2) !*My.isdep*(*x*, *b*, *z*). Первое условие разрешает правило, если дуги 6 и 7 не были использованы раннее для генерации дуги. Второе условие определяет отсутствие зависимостей между *EC*-акциями из *x* и сторожевыми условиями из *b* в существующих *AC*-зависимостях *z*. Здесь *My* – это определенный пользователем *Java*-класс и *isdep* – метод этого класса, реализующий проверку.

Артефакты трансформации графов в системе *AGG* сохраняются в специальном формате, основанном на *XML* (*GGX*-формате). Для того, чтобы использовать систему *AGG* в средствах проектирования ФБ, разработаны два конвертора из стандартного *XML*-представления ФБ в *GGX*-формат и обратно. Эти конверторы могут импортировать ФБ в *AGG*, вызывать рефакторинг и затем экспортировать результаты обратно в *XML*. Данные конверторы помогают ин-

тегрировать рефакторинг в средства поддержки проектирования программного обеспечения на основе стандарта IEC 61499.

Другой подход к реализации рефакторинга диаграмм *ECC* состоит в программной реализации правил трансформации в рамках инструментальных средств поддержки проектирования на основе IEC 61499. Например, несколько открытых (*open source*) проектов, включая 4DIAC-IDE [73] и FBench [130], могут упростить внедрение рефакторинга в практику проектирования управляющих систем на основе IEC 61499. При этом может быть разработан дружественный пользовательский интерфейс для применения правил трансформации с возможностью просмотра результатов трансформации и их принятия/неприятия.

6.7. Оценка системы рефакторинга

Визуальное, основанное на графах представление правил и трансформируемых систем – наиболее интуитивно понятная форма представления, которая позволяет избежать ошибок проектирования. Однако основное преимущество использования графовых трансформаций для рефакторинга заключается в том, что свойства систем трансформации графов хорошо изучены теоретически. Поскольку система рефакторинга является частным случаем систем трансформации (перезаписи) графов, то многие существующие теоретические результаты напрямую применимы и к ней.

Более того, системы перезаписи графов можно отнести к общему классу продукционных систем (или систем продукционных правил). Продукционная модель представления знаний является самой популярной моделью среди всех моделей представления знаний. Продукционное правило имеет вид «*ЕСЛИ <Условие>*, *ТО <Действие>*». Системы перезаписи графов наследуют многие преимущества и недостатки продукционных систем. Преимущества продукционных систем следующие: 1) модульность; 2) легкость модификации и реконфигурации; 3) выразительность; 4) простота и интуитивная понятность; 5) визуальное представление (только для систем перезаписи графов); 6) отсутствие необходимости в процессе программирования (разрабатываются и изменяются только правила, сама программа-интерпретатор не изменяется). Однако продукционные системы имеют и ряд недостатков. Недостаток, присущий всем продукционным системам: трудность проверки набора правил на полноту и непротиворечивость. Системы перезаписи графов вносят дополнительные пункты в проблемы продукционных систем.

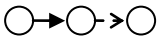
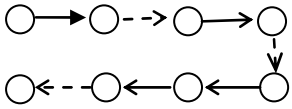
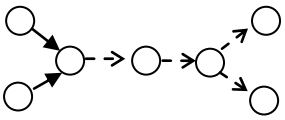
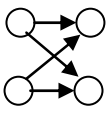
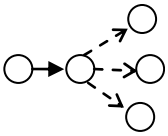
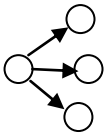
При разработке системы рефакторинга представляют интерес такие ее свойства, как сложность, полнота, корректность и конфлюентность.

Вычислительная сложность метода перезаписи графов в общем случае является высокой, поскольку основывается на задаче поиска изоморфных подграфов, которая, как известно, является *NP*-полной [149]. Однако в нашем случае подграфы в левых частях правил являются достаточно маленькими. Кроме того, метки вершин и дуг значительно сокращают пространство поиска изоморфных подграфов.

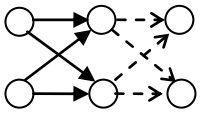
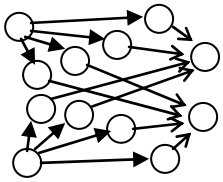
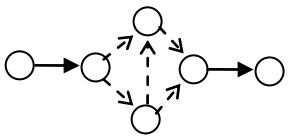
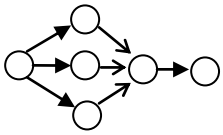
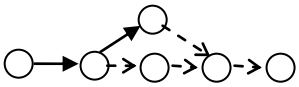
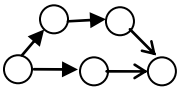
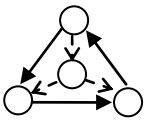
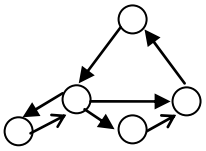
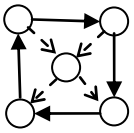
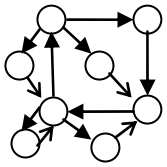
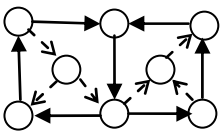
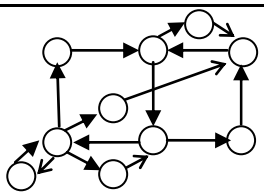
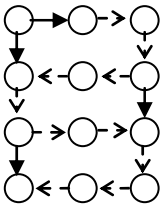
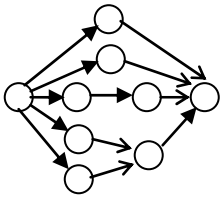
Основным методом для оценки системы рефакторинга был избран экспериментальный путь, основанный на тестировании. Система рефакторинга была протестирована с использованием более чем 30 *ECC*, представляющих различные типовые структуры. Краткие результаты некоторых тестов представлены в табл. 6.1. Испытания проводились на компьютере с процессором *Mobile Intel Pentium M*745, имеющим тактовую частоту 1,8 *GHz*, объем оперативной памяти компьютера – 1 Гб.

Таблица 6.1

Некоторые из тестов системы рефакторинга и их оценка

Название теста и краткое описание	Исходная *ECC*	Результирующая *ECC*	$T_{ср}$, ms	$P_{ср}$
ECCC-chain (последовательность из *E*-дуги и последующих *C*-дуг. Используется зависимость условия от *EC*-акции)			11,2	13
Dotted_line (последовательность из *E*-дуги, за которой следуют вперемежку *C*- и *T*-дуги)			14,5	33
Common_Chain (общая цепь из двух *C*-дуг используется для «проводки сигнала» от двух различных *E*-дуг)			13,5	29
Tuft («Куст». Включает *E*-дугу, из конца которой исходят *C*-дуги)			10,4	14

Окончание табл. 6.1

Название теста и краткое описание	Исходная *ECC*	Результирующая *ECC*	$T_{ср}$, ms	$P_{ср}$
Each_to_Each (Включает «слои» состояний. Каждое состояние предыдущего слоя связано с каждым состоянием последующего слоя с использованием сначала *E*-дуг, а затем *C*-дуг)			21,5	28
Irregular1 (нерегулярная структура)			27	35
c_Irregular1 (нерегулярная структура, включает состояние, из которого исходит как *E*-дуга, так и *C*-дуга)			23	32
Triangle («Треугольник». Замкнутая по *E*-дугам структура)			14,7	15
Square («Квадрат». Замкнутая по *E*-дугам структура)			23,8	35
Two_Square («Два квадрата»)			35,2	49
Linear2 («Линейная *C*-цепь с *E*-перекрытиями»)			70,6	117

Данные тесты были созданы эмпирически, на основе базовых топологий графов, таких как линейная последовательность, разветв-

ление, соединение, связь «каждый с каждым» и т.д. При составлении тестов исключались похожие топологии. Некоторые из тестов представляют регулярные структуры в алгебре графов. Данная алгебра позволяет строить сложные структуры на основе более простых. Однако тестировались и иррегулярные структуры. Оказалось, что они в вычислительном плане более сложные.

Для каждого теста было выполнено по три прогона. В табл. 6.1 представлено среднее время выполнения теста ($T_{ср}$) и среднее число выполненных правил ($P_{ср}$). Как можно видеть из данной таблицы, рефакторинг в системе *AGG* довольно медленный, что можно объяснить тем, что *AGG* реализована на *Java* и была создана для использования в исследовательской работе. Реализация системы перезаписи графов для рефакторинга на языке *C/C++* может дать 20-кратный выигрыш в скорости выполнения. Это может сделать систему рефакторинга пригодной для практического использования и ее включения в какую-либо интегрированную среду разработки.

Для оценки зависимости производительности системы рефакторинга от размерности *ECC* был создан набор тестов с различной размерностью. В качестве базовых тестов были выбраны: 1) тест *Linear*, представляющий линейную структуру, состоящую из *E*-дуги, за которой следует последовательность из *C*-дуг (рис. 6.1); 2) тест *Linear2*, представляющий линейную структуру из *C*-дуг с перекрытиями из *E*-дуг (рис. 6.22).

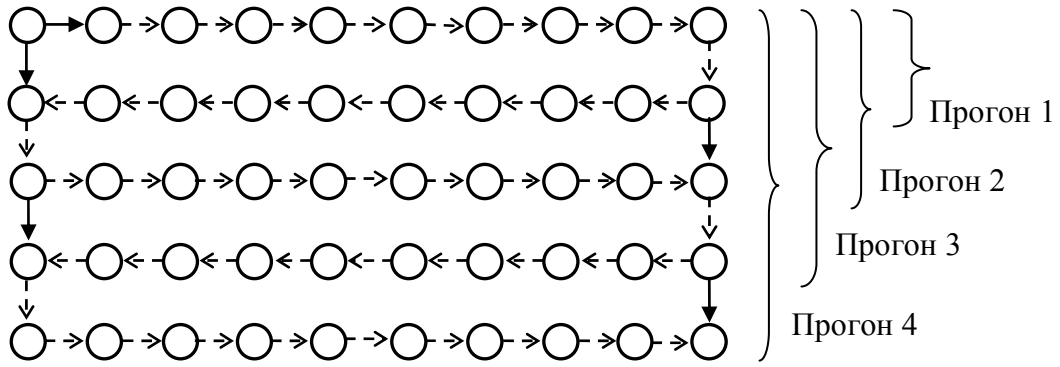


Рис. 6.22. Тест *Linear2*, используемый для оценки производительности

Как видно из рис. 6.22, трансформируемая структура представляет собой последовательность *EC*-состояний, каждое *k*-е состояние которой связано с (*k*+1)-м состоянием этой цепочки *C*-дугой, где $k = 2, 3, \ldots$ Первое состояние связано со вторым с помощью *E*-дуги. Кроме того, каждое $(10 \cdot k+1)$-е состояние цепочки связано с $(10 \cdot k + 20)$-м состоянием этой цепочки с помощью *E*-дуги. Число состояний в цепочке может варьироваться, чем и определяется сложность

теста. Однако, чтобы обеспечить равнозначность тестов в плане их структурной схожести, будем выбирать тесты, содержащие подцепочки с числом состояний, кратным 10, причем первый тест будет содержать 20 состояний.

Следует отметить, что, несмотря на то, что тест *Linear2* визуально структурно незначительно сложнее теста *Linear*, сложность трансформации структуры теста *Linear2* намного выше, чем сложность обработки чисто линейной структуры теста *Linear*. Набор применяемых правил для трансформации также более обширен. Сложность обработки объясняется наличием узловых вершин-соединителей. Это увеличивает число возможных путей в обрабатываемой структуре.

Обобщенные результаты трех прогонов для теста *Linear* приведены в табл. 6.2, а для теста *Linear2* – в табл. 6.3.

Таблица 6.2

Обобщенные результаты прогона теста *Linear*

N	$P_{ср}$	$T_{ср}$, *ms*	$TP_{ср}$, *ms*	$TPN_{ср}$, *ms*	*DT*	*PDT*, %
3	8	9283,3	1160,4	144,9	729,5	7,9
5	18	10588,3	588,2	111,6	837,7	7,9
10	43	14884,7	346,2	75,5	782,8	5,3
20	93	24388,3	262,3	65,1	668,5	2,7
30	143	41039,0	287,0	69,3	1942,6	4,7
40	193	58874,7	305,1	75,8	4127,5	7,0
50	243	87663,3	360,8	91,9	5871,3	6,7

Таблица 6.3

Обобщенные результаты прогона теста *Linear2*

N	$P_{ср}$	$T_{ср}$, *ms*	$TP_{ср}$, *ms*	$TPN_{ср}$, *ms*	*DT*	*PDT*, %
20	94	25813,7	274,6	69,5	1767,8	6,8
30	239	142151,0	594,8	157,7	11088,4	7,8
40	439	547464,0	1247,0	349,5	68883,3	12,6
50	696	1251696,3	1798,4	542,6	71474,0	5,7

В приведенных таблицах приняты следующие обозначения:

N – число состояний *ECC* (размерность *ECC*);

$P_{\text{ñð}} = (\sum_{i=1}^{3} P_i)/3$ – среднее число выполненных правил, где P_i – число правил, положительно оцененных (*true*) и выполненных в *i*-м прогоне;

$T_{\text{ñð}} = (\sum_{i=1}^{3} T_i)/3$ – среднее время выполнения теста, где T_i – время выполнения *i*-го прогона теста (в миллисекундах);

$TP_{\text{ñð}} = (\sum_{i=1}^{3} TP_i)/3$ – среднее время выполнения правила, где $TP_i = T_i/P_i$ – среднее время выполнения правила (вернее, среднее время, в течение которого выполняется одно правило. Это время включает как время выполнения правила, так долю времени, приходящуюся на непроизводительную работу – оценку ложных правил);

$TPN_{\text{ñð}} = (\sum_{i=1}^{3} TPN_i)/3$ – среднее время оценки/выполнения правила, где $TPN_i = T_i/PN_i$ – среднее время оценки/выполнения правила (учитываются как истинные, так и ложные правила), где $PN_i = P_i + N_i$ – общее число оцененных правил (как положительно, так и отрицательно);

DT – стандартное отклонение для времени выполнения теста;

$PDT = 100 \cdot DT/T_{\text{ср}}$ – процент стандартного отклонения к среднему значению.

Графическая интерпретация результатов представлена ниже. Зависимости среднего времени выполнения тестов *Linear* и *Linear2* от числа состояний диаграммы *ECC* приведены на рис. 6.23. Как можно видеть из данного рисунка, для теста *Linear* характерна линейная, слабо растущая функция, а для теста *Linear2* – полиномиальная.

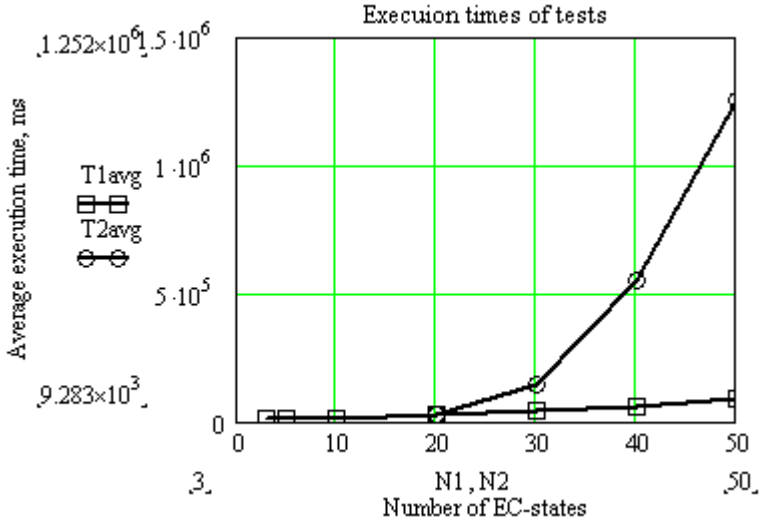


Рис. 6.23. Зависимости среднего времени выполнения тестов *Linear* (T1Avg) и *Linear2* (T2Avg) от числа состояний диаграммы *ECC*

Зависимости средних интервалов времени (разных видов), в которых выполняется одно правило, от числа состояний диаграммы *ECC* приведены на рис. 6.24.

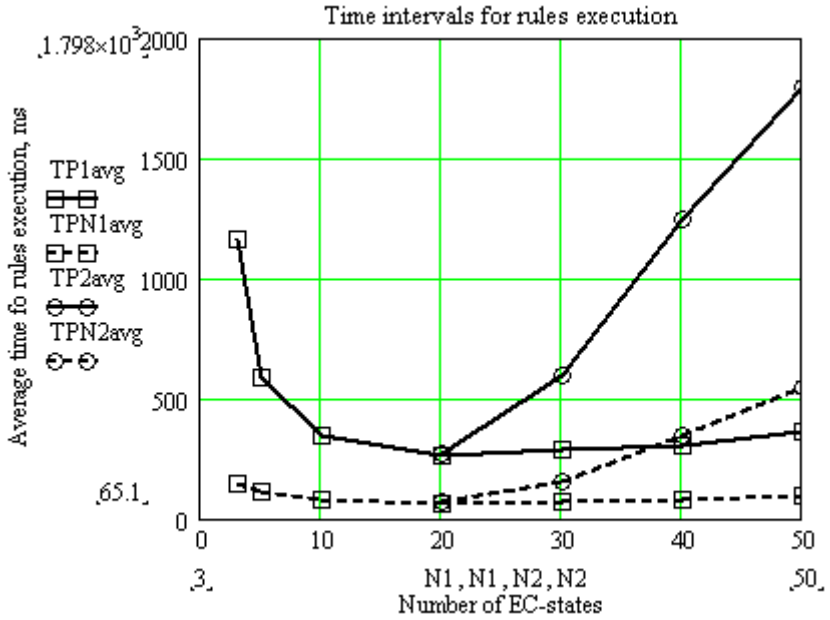


Рис. 6.24. Зависимости средних интервалов времени,
в которых выполняется одно правило,
от числа состояний диаграммы *ECC*

На данном рисунке приняты следующие обозначения зависимых величин:

TP1avg – средний интервал времени, в котором выполняется одно (разрешенное) правило в тесте *Linear* (иными словами – среднее время выполнения правила в тесте *Linear*);

TP2avg – средний интервал времени, в котором выполняется одно (разрешенное) правило в тесте *Linear2* (иными словами – среднее время выполнения правила в тесте *Linear2*);

TPN1avg – средний интервал времени, в котором выполняется одно (разрешенное) правило или оценивается одно (неразрешенное) правило в тесте *Linear* (иными словами – среднее время обработки правила в тесте *Linear*);

TPN2avg – средний интервал времени, в котором выполняется одно (разрешенное) правило или оценивается одно (неразрешенное) правило в тесте *Linear2* (иными словами – среднее время обработки правила в тесте *Linear2*).

Как видно из рис. 6.24, для теста *Linear* в графиках зависимостей *TP1avg* и *TPN1avg* имеется минимум – сначала среднее время выполнения правил падает, а потом растет. Объясняться это может тем, что имеется постоянная, достаточно большая величина времени, которая необходима для инициализации системы. Сначала при малом числе состояний (а значит, и обрабатывающих правил) ее

(удельное) влияние большое, потом с ростом числа состояний оно ослабевает, и среднее время выполнения правил уменьшается. Однако с последующим ростом числа состояний наблюдается рост среднего времени выполнения правил за счет увеличения размерности анализируемого графа, а значит, и времени поиска изоморфных подграфов.

Для теста *Linear2* такой закономерности (минимума) не видно из-за невозможности построить данный тест для малого числа состояний (шаг дискретизации равен 10). Однако можно наблюдать, что среднее время выполнения правил для теста *Linear2* имеет почти линейный характер. Также можно не сомневаться, что линейный характер имеют и соответствующие зависимости для теста *Linear1* за отметкой $N > 50$ (на графике не показано).

Зависимости числа выполненных правил для тестов *Linear* и *Linear2* от числа состояний диаграммы *ECC* приведены на рис. 6.25. Как можно видеть из рисунка, для теста *Linear* характерна линейная функция, а для теста *Linear2* – полиномиальная.

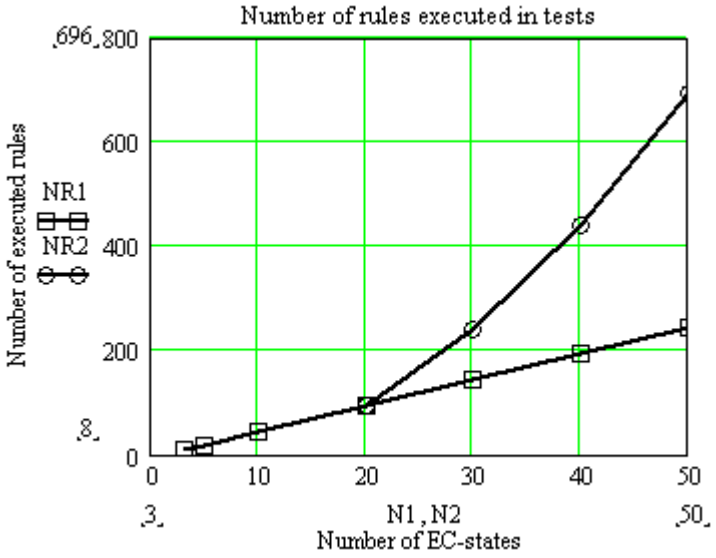


Рис. 6.25. Зависимости числа выполненных правил для тестов *Linear* (*NR*1) и *Linear2* (*NR*2) от числа состояний диаграммы *ECC*

Завершаемость (*termination*) работы системы перезаписи графов является в общем случае неразрешимой проблемой (однако завершаемость может быть обеспечена, если правила трансформации удовлетворяют некоторым критериям [124]). Таким образом, проблема завершаемости рефакторинга в общем случае также неразрешима. На практике может встретиться ситуация, когда рефакторинг выполняется продолжительное время и невозможно предсказать за-

ранее, будет ли вообще когда-либо закончен этот процесс. Однако все рассмотренные выше тесты завершились. Также была произведена проверка на соответствие системы рефакторинга критериям завершаемости с использованием *AGG*. В результате этой проверки получено, что данные критерии не удовлетворяются. Действительно, возможен бесконечный вывод на графах в случае использования циклических *CT*-сетей в структуре *ECC*, но как было отмечено выше, данные *ECC* не рассматриваются. Следует также отметить, что анализ, реализованный в системе *AGG*, не учитывает скрытые зависимости между атрибутами, выраженные в *Java*-классах, определенных пользователем.

Полнота системы правил означает, что гарантируется достижение цели рефакторинга для любых исходных *ECC*. Представленная выше система рефакторинга создавалась эмпирически. При этом свойство полноты формально не доказывалось. Однако все представленные выше тесты подтверждают данное свойство.

Суть *конфлюенции* состоит в том, что обеспечивается один и тот же результат при различном порядке применения правил. В данной работе за основу были взяты результаты [153], согласно которым система трансформации типизированных атрибутных графов конфлюентна, если все критические пары правил конфлюентны. По определению [153, 179] критическая пара правил (p1, p2) существует, если применение правила p1 деактивирует правило p2, и наоборот. Система *AGG* может производить анализ критических пар для заданной системы правил. Множество обнаруженных критических пар точно представляет все потенциальные конфликты. С помощью средств *AGG* было обнаружено и исправлено несколько ошибок в правилах рефакторинга. Однако процесс анализа критических пар в системе *AGG* является медленным вследствие вычислительной сложности. Для подтверждения свойства конфлюентности представленные выше тесты прогонялись несколько раз, при этом всегда достигался один и тот же результат. Таким образом, свойство конфлюентности определялось экспериментальным путем.

Основные положения данного раздела опубликованы в работах [41, 118, 246].

7. Семантический анализ проектов IEC 61499 на основе Web-онтологий

В данном разделе рассматривается онтологический подход к описанию и семантическому анализу систем управления на основе стандарта IEC 61499 с использованием дескриптивной логики (ДЛ) [224] и логики хорновских дизъюнктов (*Horn logic*).

В качестве источников для построения онтологии стандарта IEC 61499 в данной работе использовались: 1) текст стандарта IEC 61499 (часть 1) [163]; 2) синтаксис текстового языка ФБ (приложение *B* Части 1 стандарта IEC 61499 [163]); 3) *DTD*-блок для представления синтаксиса *XML*-документов, описывающих ФБ (Часть 2 стандарта IEC 61499 [164]); 4) примеры систем ФБ из интегрированной среды проектирования *FBDK* [134]. В онтологию закладываются три вида знаний: 1) знания о синтаксисе языка ФБ; 2) знания о семантических свойствах и ограничениях описаний систем ФБ, получаемые из анализа текста стандарта; 3) знания о глубинных взаимосвязях между объектами описаний, полученные в результате дополнительного анализа систем ФБ.

При разработке онтологии ФБ попутно ставилась задача выбора и *минимизации* числа *базовых* классов, под которыми в данном случае понимались классы, экземпляры которых создавались бы автоматическим транслятором из исходных *XML*-описаний ФБ. Данный подход является рациональным, поскольку транслятор в данном случае работает только с фактографической информацией (на уровне синтаксических знаний), а семантический анализ возложен исключительно на систему рассуждений. Это позволяет значительно облегчить разработку транслятора. В то же время на транслятор может быть возложена задача поиска и выявления глубинных знаний, которые невозможно выполнить в системе рассуждений. Например, это может быть связано с анализом алгоритмов и их взаимосвязей с переменными. Данные зависимости не указываются в исходном *XML*-описании – алгоритмы представляются лишь в виде текстовой строки, без всякой дополнительной структуризации.

Для описания разработанной онтологии ФБ будем использовать следующие средства: 1) интенсиональные семантические сети [68, 213]; 2) описания концептов (классов) и ролей (отношений, свойств) с использованием аксиом в виде формул ДЛ; 3) правила языка *SWRL*. В семантических сетях концепты будут изображаться в виде в виде овалов, а роли – в виде стрелок. Концептам даются

самоопределяемые имена. Свойства онтологии типа *Data Properties* не рассматриваются. В соответствии с общепринятой практикой имена классов онтологии будем писать с большой буквы, а имена свойств – с маленькой.

7.1. Онтология функциональных блоков

7.1.1. Использование онтологии для решения проблем синтаксиса ФБ

В онтологии синтаксическое описание системы ФБ тесно переплетается с семантическим описанием. Онтологические средства описывают и то, и другое одинаковыми методами. Но попытаемся условно разделить эти две сферы. Будем относить к семантическому описанию ФБ то, что явно не описывается синтаксисом языка ФБ.

Существует три формы представления ФБ: графическая, текстовая и в виде *XML*-документа. Соответственно, существуют три языка ФБ: графический, текстовый и *XML*-базированный язык (это множество *XML*-документов, представляющих системы ФБ). По идее, все три языка должны быть эквивалентны.

Существует два стандартных определения синтаксиса языка ФБ: 1) с помощью расширенных форм Бэкуса-Наура (РБНФ) (Часть 1 стандарта, прил. В) [163]; 2) с помощью *DTD* (Часть 2 стандарта) [164]. Первый синтаксис определяет текстовый язык описания ФБ, а второй синтаксис описывает структуру *XML*-документов, представляющих системы ФБ. Существуют довольно сильные различия данных синтаксисов в описании одних и тех же языковых конструкций. Первый синтаксис, как правило, подробнее и поэтому более предпочтительный. Следует сразу заметить, что сравнение этих двух синтаксисов не является целью данного исследования. Все же некоторое сравнение будет проводиться подспудно, в рамках оценки онтологического описания синтаксических конструкций. Следует также заметить, что встречаются ситуации, когда средств второго синтаксиса (несмотря на его подробность) бывает недостаточно для полного и точного описания ситуации в системах ФБ (в синтаксическом плане). Предложенная ниже онтология ФБ довольно точно следует текстуальному синтаксису, но в ряде «узких» мест существующий синтаксис дополняется семантическими ограничениями.

Ниже будут рассматриваться те особенности синтаксиса, которые нестандартны и действительно вызывают интерес с точки зрения онтологического описания проблемных ситуаций. При описании проблемных языковых конструкций будем приводить разные

формы описания их синтаксиса: 1) *DTD*-синтаксис; 2) текстуальный синтаксис; 3) «онтологический» синтаксис. Это будет полезно для их сравнения. В целом, в синтаксическом плане онтология: 1) доопределяет синтаксис в отношении раздельного описания типов базисных, составных и сервисных интерфейсных ФБ; 2) доопределяет синтаксис *EC*-акций в плане уточнения их структуры и используемых типов событийных выходов; 3) дает точное синтаксическое описание связей (событийных, информационных, адаптерных).

7.1.2. Таксономия типов и экземпляров

В стандарте определен ряд элементов, имеющих интерфейсы. В дальнейшем будем называть их ФБ-подобными элементами, поскольку по внешнему виду они похожи на ФБ. Различаются типы ФБ-подобных элементов (класс `FB_like_type`) и их экземпляры (класс `FB_like_instance`). К типам ФБ-подобных элементов относятся: ФБ-тип (класс `FB_type`), субприложение-тип (класс `subapplication_type`), адаптер-тип (класс `adapter_type`). В соответствии со стандартом ФБ могут быть трех типов: базисные ФБ, составные ФБ и сервисные интерфейсные ФБ (СИФБ). Таксономия классов, определяющих типы ФБ-подобных элементов, приведена на рис. 7.1. В соответствии с отношением наследования все классы приведенной иерархии будут иметь свойство `has_interface`.

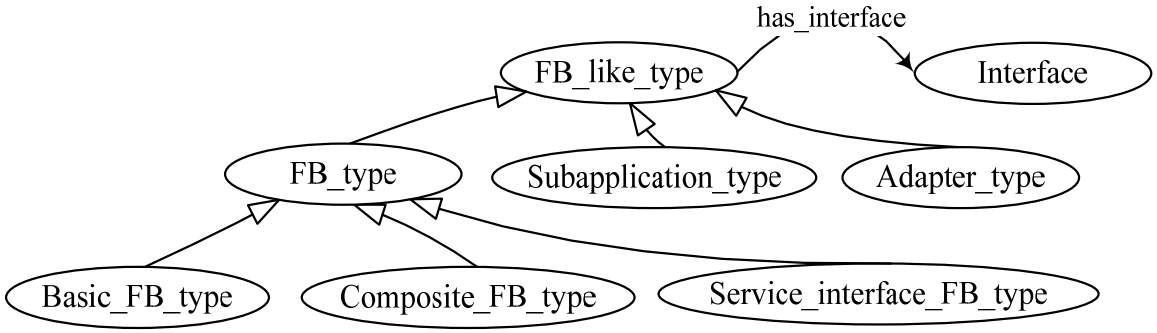


Рис. 7.1. Таксономия типов ФБ-подобных элементов

Для различения типов и экземпляров элементов к последним добавляется слово «компонентный», например, компонентный ФБ. Часто, когда из контекста понятно, что речь идет об экземпляре, данное слово опускается. Компонентные адаптеры бывают двух видов: сокеты (класс `socket`) и штекеры (класс `plug`). Таксономия классов, определяющих экземпляры ФБ-подобных элементов, приведена на рис. 7.2.

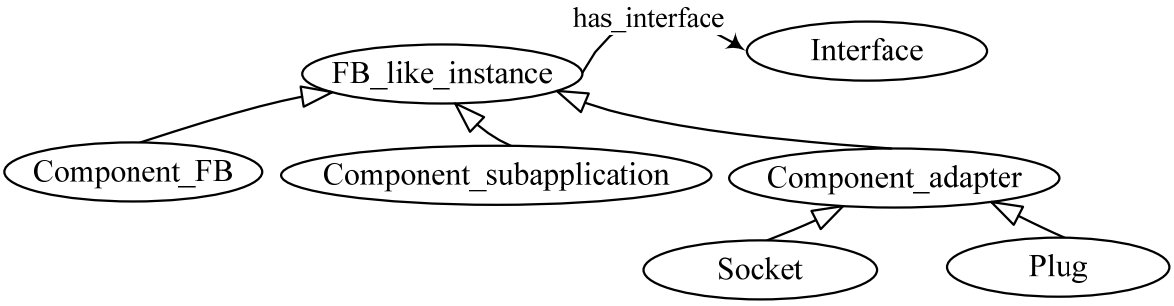


Рис. 7.2. Таксономия экземпляров ФБ-подобных элементов

7.1.3. Описание интерфейсов

Подразделение интерфейсов, а также событийных и информационных входов и выходов на классы будем производить согласно их принадлежности тому или иному типу или экземпляру, который является собственником этих интерфейсов.

На рис. 7.3 в качестве примера приведено описание интерфейса типа ФБ и адаптера.

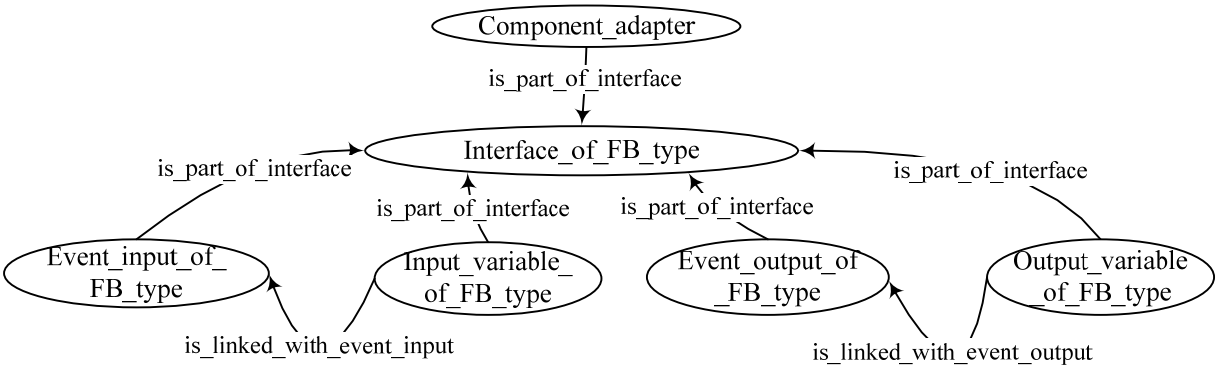


Рис. 7.3. Описание интерфейса типа ФБ и адаптера

На рис. 7.4 представлена таксономия классов событийных входов/выходов (класс `event`), а на рис. 7.5 – таксономия классов для определения переменных (класс `variable`).

Как видно из рис. 7.5 в класс `variable` включены не только интерфейсные переменные, но и внутренние переменные базисных ФБ (класс `internal_variable_of_basic_FB_type`) и переменные, используемые на устройствах или ресурсах (класс `variable_of_device_and_resource`).

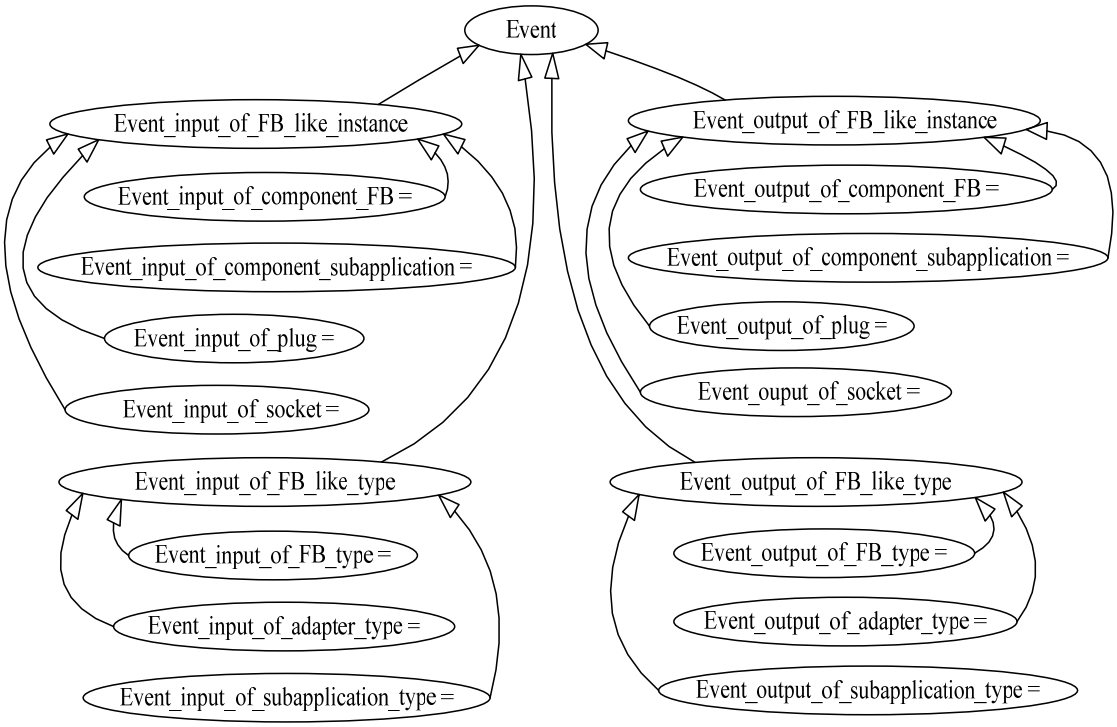


Рис. 7.4. Таксономия событийных входов/выходов

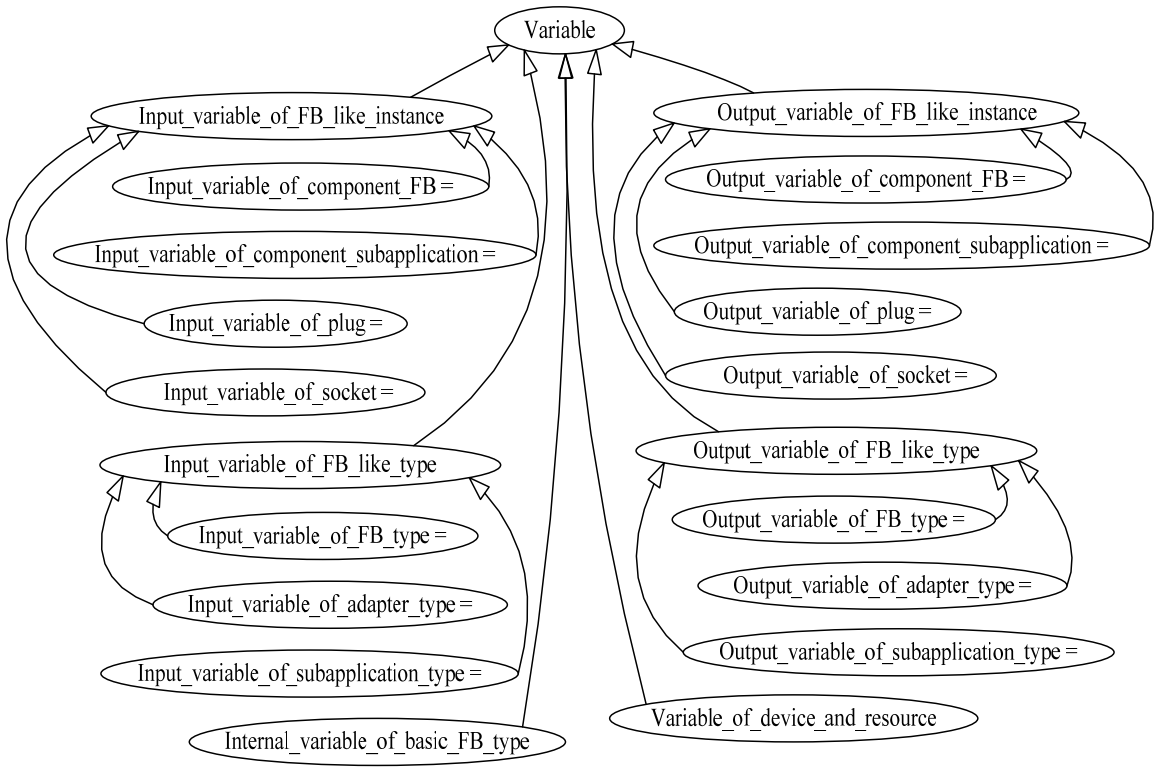


Рис. 7.5. Таксономия переменных

Для уменьшения разнообразия *базовых* классов при представлении интерфейсов сделаем их «*вычисляемыми*» с учетом их принадлежности тому или иному элементу. Так, например, класс ин-

терфейсов компонентных ФБ определим в ДЛ с использованием отношения эквивалентности следующим образом:

$$\begin{aligned} &\texttt{interface\_of\_component\_FB} \equiv \\ \equiv\ &\texttt{interface\_of\_FB\_like\_instance} \\ \sqcap\ &\exists\ \texttt{has\_interface}^{-}\texttt{.component\_FB} \end{aligned}$$

Здесь знак минус в зоне верхнего индекса отношения обозначает отношение, обратное исходному.

Для уменьшения разнообразия *базовых* классов событийных входов/выходов и переменных часть их также делается «*вычисляемой*». Пример определения событийных выходов субприложений-типов:

$$\begin{aligned} &\texttt{event\_output\_of\_subapplication\_type} \equiv \\ \equiv\ &\texttt{event\_output\_of\_FB\_like\_type} \sqcap \\ &\exists\ \texttt{is\_part\_of\_interface.(interface} \sqcap \exists \\ &\texttt{has\_interface}^{-}\texttt{. subapplication\_type)} \end{aligned}$$

На рис. 7.4 и 7.5 вычисляемые классы отмечены знаком = после имени класса.

7.1.4. Корректировка синтаксиса в отношении описания типов ФБ

Недостаток *DTD*-синтаксиса в отношении описания типов ФБ состоит в том, что *DTD* позволяет определение сервисов для базисных и составных ФБ (см. первую строку табл. 7.1). Такой «смешанный» тип ФБ не разрешен Стандартом (см. прил. В Части 1 [163]). Но и текстуальный синтаксис описывает то же самое таким же образом (см. вторую строку табл. 7.1). Была сделана попытка выправить ситуацию с помощью комментариев, но комментарии не входят в формальное определение синтаксиса языка, а потому такой подход является неформальным. Построение синтаксического анализатора строго по текстуальному синтаксису приведет к тому, что будет допускаться описание «смешанного» типа ФБ, что является ошибкой. Таким образом, в дополнение к синтаксическому анализатору необходим также и некий семантический анализатор, который будет использовать «внешнюю» информацию, заложенную в комментариях.

Для разрешения проблемной ситуации в онтологии общее понятие «Тип ФБ» было разделено на три понятия «Тип базисного ФБ», «Тип составного ФБ» и «Тип сервисного интерфейсного ФБ» с четко определенными свойствами. Данные классы понятий сдела-

ны непересекающимися. Ниже подробно рассматривается онтологическое описание всех трех типов ФБ.

Таблица 7.1

Три способа описаний типов ФБ

Синтаксис в соответствии с частью 2 стандарта (DTD) [164]	`<!ELEMENT FBType (Identification?,VersionInfo+,CompilerInfo?,InterfaceList, (BasicFB | FBNetwork)?, Service?) >`

Текстуальный синтаксис в соответствии с прил. В части 1 стандарта (РФБН) [163]	`a) fb_type_declaration ::=`
	`'FUNCTION_BLOCK' fb_type_name`
	`fb_interface_list`
	`[fb_internal_variable_list] <only for basic FB>`
	`[fb_instance_list] <only for composite FB>`
	`[plug_list]`
	`[socket_list]`
	`[fb_connection_list] <only for composite FB>`
	`[fb_ecc_declaration] <only for basic FB>`
	`{fb_algorithm_declaration} <only for basic FB>`
	`[fb_service_declaration] <only for service interface FB>`
	`'END_FUNCTION_BLOCK'`

Представление скорректированного синтаксиса с использованием онтологий	`Basic_FB_Type ⊑ FB_type ⊓ =1 has_ECC.ECC`
	`Composite_FB_Type ⊑ FB_type ⊓ =1 has_FB_network. FB_network`
	`Service_Interface_FB_Type ⊑ FB_type ⊓ =1 Service`
	`Basic_FB_Type ⊓ Composite_FB_Type ⊓ Service_Interface_FB_Type = ∅ (disjoint classes)`

7.1.5. Описание (типов) базисных ФБ

Базисные ФБ являются основными в стандарте. Они включают в свой состав машину состояний, управляемую событиями. Для ее представления используется диаграмма управления выполнением (диаграмма *ECC*). Диаграмма *ECC* состоит из *EC*-состояний (класс `EC_state`) и *EC*-переходов (класс `EC_transition`). С каждым

EC-состоянием может быть связан набор *EC*-акций (класс `EC_action`), каждая из которых может включать алгоритм и выходное событие. С *EC*-переходом связывается условие перехода, которое может состоять из входного события и сторожевого условия (класс `Guard_condition`).

Семантическая сеть для описания базисных ФБ представлена на рис. 7.6. На данной диаграмме для представления ранга отношения, определяемого в виде объединения нескольких множеств, используется рамка прямоугольника со скругленными краями, над которой надписывается символ *OR*.

Для более точного описания базисного ФБ вводится ряд дополнительных отношений между соответствующими концептами, используемых в семантическом анализе описания. Все эти отношения являются неявно вычисляемыми (на основе дополнительной обработки исходного описания) и относятся к знаниям типа 3 (см. выше). На рис. 7.6 в качестве примера представлено подобное отношение *is_used_in*.

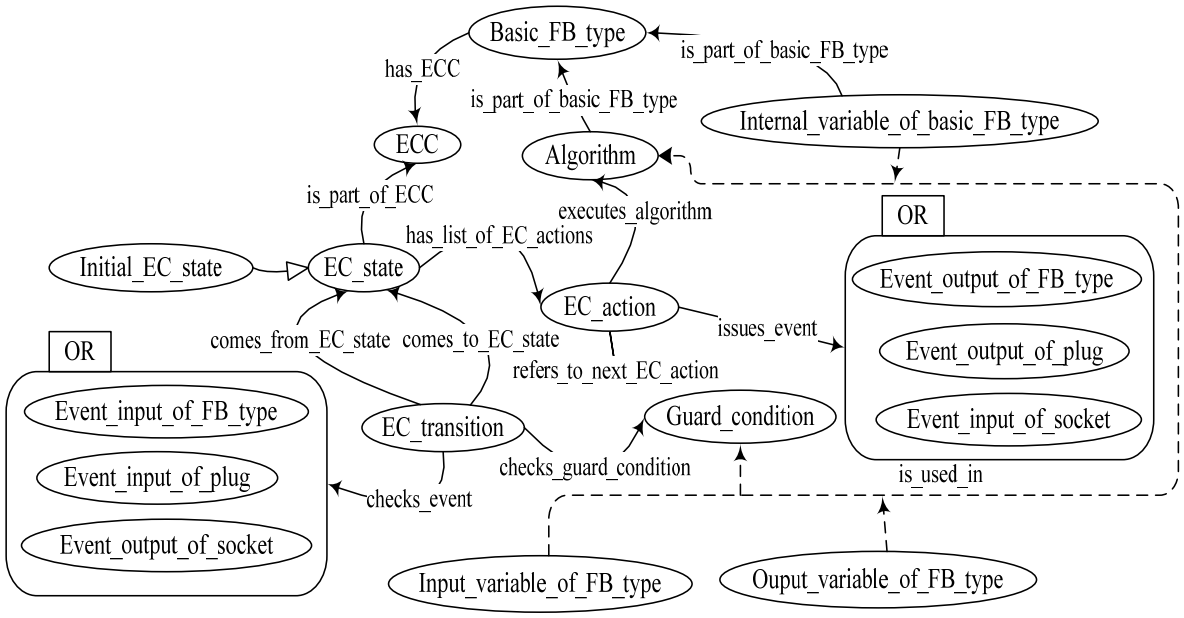


Рис. 7.6. Семантическая сеть для определения типов базисных ФБ

Смысловая интерпретация используемых в диаграмме отношений (иначе, объектных свойств) представлена в табл. 7.2. Отношение *R* и вовлеченные в него объекты обозначаются в виде тройки: *x R y*. При существовании обратного свойства оно указывается в скобках после прямого свойства. Определение классов субонтологии базисных ФБ в виде формул ДЛ представлено в табл. 7.3.

Таблица 7.2

Отношения субонтологии базисных ФБ

Имя свойства (обратное свойство)	Смысловая интерпретация
checks_event (is_checked_by)	В условии *EC*-перехода *x* проверяется значение событийного входа *y*
checks_guard_condition	В условии *EC*-перехода *x* проверяется сторожевое условие *y*
comes_from_EC_state (origi-nates_EC_transition)	*EC*-переход *x* исходит из *EC*-состояния *y*
comes_to_EC_state (finishes_EC_transition)	*EC*-переход *x* входит в *EC*-состояние *y*
executes_algorithm	EC-акция *x* включает выполнение алгоритма *y*
has_ECC	Базисный ФБ *x* содержит диаграмму *ECC y*
has_interface	Базисный ФБ *x* содержит интерфейс *y*
has_list_of_EC_actions	*EC*-состояние *x* обладает списком акций *y*
is_part_of_basic_FB_type	Объект *x* является частью (типа) базисного ФБ *y*
is_part_of_ECC	*EC*-состояние *x* является частью диаграммы *ECC y*
issues_event (is_issued_by)	*EC*-акция *x* активирует событие *y*
refers_to_next_EC_action	*EC*-акция *x* ссылается на следующую *EC*-акцию *y* в списке *EC*-акций
is_used_in	Объект *x* используется при вычислении *y*

Таблица 7.3

Определение классов субонтологии для базисных ФБ

Имя класса	Аксиомы класса
Алгоритм	algorithm ⊑ =1 is_part_of_basic_FB_type. basic_FB_type
EC-акция	EC_action ⊑ (=1 exe-cutes_algorithm.algorithm ⊔ =1 is-sues_event.(event_input_of_socket ⊔ event_output_of_FB_type ⊔ event_output_of_plug)) ⊓ ≤1 re-fers_to_next_EC_action. EC_action
EC-состояние	EC_state ⊑ =1 is_part_of_ECC.ECC ⊓ ≤ 1 has_list_of_EC_actions.EC_action

Окончание табл. 7.3

Имя класса	Аксиомы класса
EC-переход	EC_transition ⊑ (=1 checks_event.(event_input_of_FB_type ⊔ event_input_of_plug ⊔ event_output_of_socket) ⊔ =1 checks_guard_condition.guard_condition) ⊓ =1 comes_from_EC_state.EC_state ⊓ =1 comes_to_EC_state.EC_state
Внутренняя переменная базисного ФБ	internal_variable_of_basic_FB_type ⊑ variable ⊓ =1 is_part_of_basic_FB_type.basic_FB_type

Кратко рассмотрим недостатки существующих описаний базисных ФБ. Общий недостаток *DTD*-синтаксиса в отношении описания *EC*-акций состоит в том, что *DTD* допускает «пустые» *EC*-акции без соответствующих алгоритма и выходного сигнала, но в соответствии со стандартом какой-нибудь один из этих компонентов должен присутствовать. Текстуальный синтаксис лишен этого недостатка, но в то же время текстуальный синтаксис описывает событийный выход *event_output_name* в *EC*-акции довольно грубо – на уровне имен. На первый взгляд кажется вполне очевидным, что *event_output_name* определяет имя событийного выхода базисного ФБ. Однако следует учесть (о чем часто забывают), что базисный ФБ может содержать (в интерфейсе) как сокеты, так и штекеры. И сигнал, вырабатываемый *EC*-акцией, может посылаться не на выход собственно ФБ, а на входы-выходы данных адаптеров, причем не на все. Например, сигнал не может быть послан на событийный выход сокета.

Предлагаемое онтологическое описание «выправляет» указанную выше ситуацию. В данном описании уточняется, что в качестве событийного выхода могут использоваться: а) событийный выход ФБ-типа; б) событийный вход сокета; в) событийный выход штекера. Это можно отнести к семантическим ограничениям, отсутствующим в описании синтаксиса. Все три вида синтаксических описаний: на основе *DTD*, в виде РБНФ, с использованием онтологий, представлены в табл. 7.4.

Таблица 7.4

Три способа описаний типов *EC*-акций

Синтаксис в соответствии с частью 2 стандарта (DTD) [228]	`<!ELEMENT ECAction EMPTY>`
	`<!ATTLIST ECAction`
	`  Algorithm CDATA #IMPLIED`
	`  Output CDATA #IMPLIED >`

Текстуальный синтаксис в соответствии с прил. В части 1 стандарта (РФБН) [163]	`ec_action ::= algorithm_name | '->' event_output_name` `| algorithm_name '->' event_output_name`
Представление скорректированного синтаксиса с использованием онтологий	`EC_action ⊑ (=1 executes_algorithm.Algorithm) ⊔ (=1 issues_event.(Event_input_of_socket ⊔ Event_output_of_FB_type ⊔ Event_output_of_plug)) ⊓ ≤1 refers_to_next_EC_action.EC_action`

7.1.6. Описание связей

Поскольку язык ФБ является структурно-ориентированным языком, то связи между элементами играют большую роль в представлении систем управления. Сеть ФБ (класс `FB_network`) является важнейшей составной частью таких артефактов проектирования, как составной ФБ, субприложение, приложение, устройство, тип устройства, ресурс, тип ресурса.

Общий недостаток *DTD*-синтаксиса в отношении описания связей состоит в том, что из-за своей «грубости» описания *DTD* допускает связи практически между любыми «контактами» сети ФБ без всяких ограничений. Это дает возможность задания заведомо бессмысленных связей, например, можно описать связь из входа одного компонентного ФБ на вход другого. Описания событийных, информационных и адаптерных связей являются одинаковым. Кроме того, связи ФБ и субприложений не различаются. Текстуальный синтаксис более точен и более насыщен. Он определяет различающиеся подробные описания для событийных, информационных и адаптерных связей, а также связей субприложений. В то же время следует указать на некоторые неточности (или, скорее, даже ошибки) в описании связей в текстуальном синтаксисе, что будет проиллюстрировано ниже на примере описания событийных связей в ФБ (табл. 7.5)

Таблица 7.5

Три способа описаний событийных связей в ФБ

Синтаксис в соответствии с частью 2 стандарта (DTD) [164]	<!ELEMENT EventConnections (Connection+)> <!ELEMENT Connection EMPTY> <!ATTLIST Connection Source CDATA #REQUIRED Destination CDATA #REQUIRED Comment CDATA #IMPLIED dx1 CDATA #IMPLIED dx2 CDATA #IMPLIED dy CDATA #IMPLIED >
Текстуальный синтаксис в соответствии с прил. В части 1 стандарта (РФБН) [163]	event_conn ::= ((fb_instance_name '.' event_output_name) | (plug_name '.' event_input_name)) 'TO' ((fb_instance_name '.' event_input_name) | (plug_name '.' event_output_name)) ';' | event_input_name 'TO' ((fb_instance_name '.' event_input_name) | (plug_name '.' event_output_name)) ';' | ((fb_instance_name '.' event_output_name) | (plug_name '.' event_input_name)) 'TO' event_output_name ';'
Представление скорректированного синтаксиса с использова-нием онтологий	Syntax_right_event_connection_of_FB ≡ Event_connection ⨅ =1 comes_from_event.(Event_input_of_FB_type ⨆ Event_input_of_plug ⨆ Event_output_of_component_FB ⨆ Event_output_of_socket) ⨅ =1 comes_to_event.(Event_input_of_component_ FB ⨆ Event_input_of_socket ⨆ Event_output_of_FB_type ⨆ Event_output_of_plug)

Рассмотрим подробнее, какие виды связей описываются текстуальным синтаксисом в отношении событийных дуг в ФБ. Результаты представлены в табл. 7.6. В данной таблице представлены все разумные виды связей. Закрашенные строки соответствуют связям, которые «покрываются» текстуальным синтаксисом.

Таблица 7.6

Покрытие типов связей («источник-приемник») текстуальным синтаксисом описания событийных связей

Источник связи (тип)	Приемник связи (тип)
`event_input_name`	`fb_instance_name '.' event_input_name`
То же	`socket_name'.' event_input_name`
То же	`event_output_name`
То же	`plug_name '.' event_output_name`
`plug_name '.' event_input_name`	`fb_instance_name'.' event_input_name`
То же	`socket_name'.' event_input_name`
То же	`event_output_name`
То же	`plug_name '.' event_output_name`
`fb_instance_name '.' event_output_name`	`fb_instance_name'.' event_input_name`
То же	`socket_name'.' event_input_name`
То же	`event_output_name`
То же	`plug_name '.' event_output_name`
`socket_name '.' event_output_name`	`fb_instance_name'.' event_input_name`
То же	`socket_name'.' event_input_name`
То же	`event_output_name`
То же	`plug_name '.' event_output_name`

Из табл. 7.6 видно, что текстуальный синтаксис покрывает только 50 % всех типов связей. Можно предположить, что остальные связи, соответствующие не закрашенным строкам, не разреше-

ны стандартом. Однако это далеко не так. Например, из табл. 7.6 видно, что сокет вообще ни с чем не может быть связан. Но если рассмотреть рис. 18 первой части стандарта [163], то можно заметить, что сокет может иметь как входные, так и выходные линии. Более того, в параграфе 5.5.3 первой части стандарта явно сказано: «*…Inputs and outputs of a socket shall be used within its function block type declaration in the same manner as outputs and inputs of the function block, respectively…*», т.е. сокет (да и штекер, см. стандарт) используется в описании типа ФБ на правах обычного компонентного ФБ.

В качестве второго примера рассмотрим еще один неразрешенный синтаксисом вид связей «*event_input_name → event_output_name*». Это прямая событийная связь со входа на выход ФБ-типа. По поводу этого типа связи в параграфе 5.3.1 (d) первой части стандарта сказано: «*...Each event output of the composite function block is connected from exactly one event output of exactly one component function block, or from exactly one event input of the composite function block, with the exception that the graphical shorthand for event merging shown in Figure A.1 may be employed…*». Таким образом, в тексте стандарта эта связь разрешена.

Приведенное в табл. 7.5 онтологическое описание событийных связей в ФБ свободно от отмеченных недостатков текстуального синтаксиса. На рис. 7.7 схематично представлены класс `Syntax_right_event_connection_of_FB` , его домен и ранг, а также связи с окружением. Как видно из рис. 7.7, на синтаксически правильную событийную связь для составных ФБ накладываются более сильные ограничения, касающиеся возможных типов источников и приемников дуги.

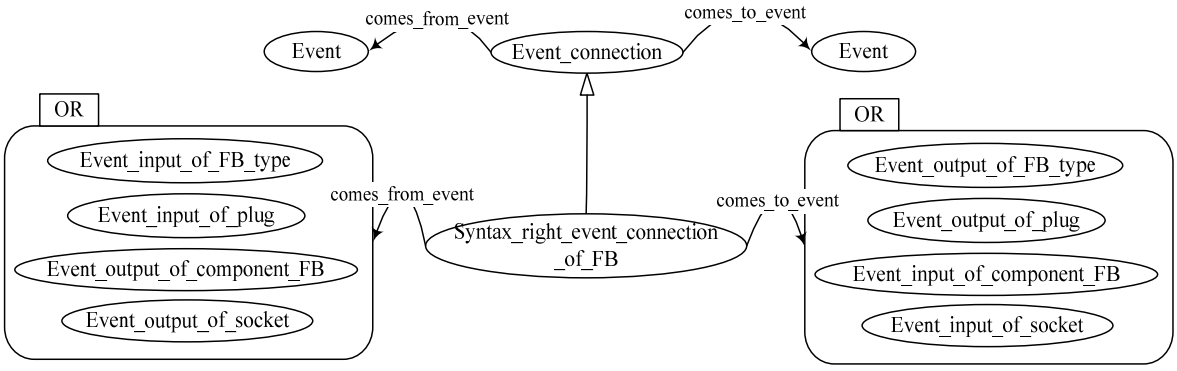


Рис. 7.7. Отношение между событийной связью и ее синтаксически правильным вариантом для составных ФБ

Похожие проблемы в текстуальном синтаксисе имеются и с описанием информационных связей (*data connection*) в ФБ.

По крайней мере, там также отсутствуют связи, которые имеют отношение к сокетам. Таким образом, Рабочей группе, ответственной за сопровождение стандарта, следует подкорректировать описание текстуального синтаксиса.

Классификация связей (отображенная с помощью системы *Protégé*) приведена на рис. 7.8. Как видно из рис. 7.8, все связи делятся на три больших класса: событийные, информационные и адаптерные связи. Особо выделяются так называемые «синтаксически правильные» классы для связей.

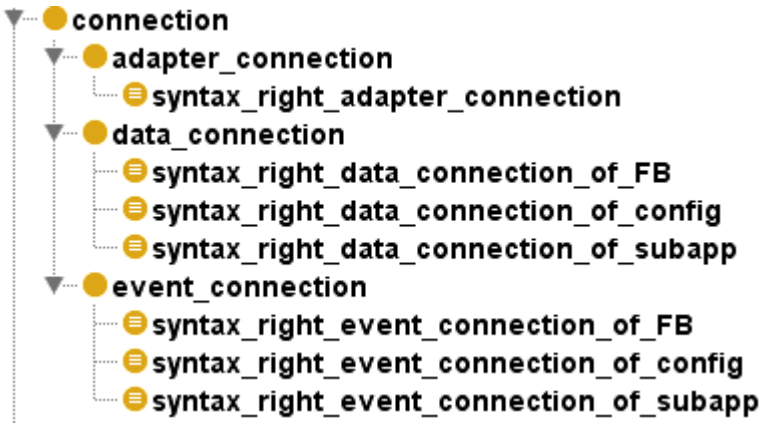


Рис. 7.8. Таксономия связей (вид в Protégé)

Ниже приводятся показательные примеры определений классов связей на основе ДЛ. Синтаксически правильные событийные связи для субприложения определяются почти так же, как и для ФБ (см. табл. 7.5), но при этом класс `event_input_of_component_FB` необходимо заменить на класс `event_input_of_component_subapplication`.

В случае конфигураций на событийные связи накладываются еще большие ограничения на типы источников и приемников связей. В этом случае синтаксически правильные событийные связи определяются как

```
Syntax_right_event_connection_of_config ≡
Event_connection
⊓ =1
comes_from_event.Event_output_of_component_FB
⊓ =1
comes_to_event.Event_input_of_component_FB
```

Синтаксически правильные информационные связи для составного ФБ определяются следующим образом:

```
   Syntax_right_data_connection_of_FB ≡ Da-
ta_connection
   ⊓ =1
comes_from_variable.(Input_variable_of_FB_type ⊔
Input_variable_of_plug ⊔ Out-
put_variable_of_component_FB ⊔ Out-
put_variable_of_socket)
   ⊓ =1
comes_to_variable.(Input_variable_of_component_FB
⊔ Input_variable _of_socket ⊔ Out-
put_variable_of_FB_type ⊔ Out-
put_variable_of_plug)
```

Синтаксически правильные адаптерные связи определяются следующей аксиомой класса:

```
   Syntax_right_adapter_connection ≡ Adapt-
er_connection
   ⊓ =1
comes_from_adapter.(Plug_of_FB_like_instance ⊔
Socket)
   ⊓ =1 comes_to_adapter.(Plug ⊔ Sock-
et_of_FB_like_instance)
```

7.1.7. Описание (типов) составных ФБ

Составной ФБ определяется входящей в его состав сетью ФБ, включающей компонентные ФБ и субприложения, а также связи между ними.

На рис. 7.9 показана семантическая сеть для определения составных ФБ.

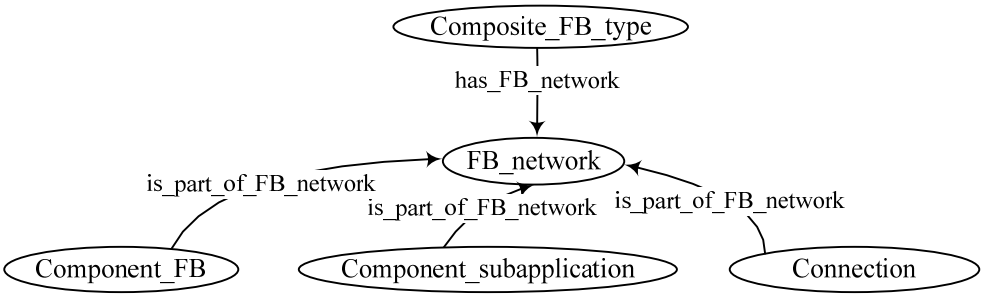


Рис. 7.9. Семантическая сеть для определения типов составных ФБ

7.1.8. Описание (типов) сервисных интерфейсных ФБ

Особенностью онтологического описания СИФБ является широкое использование *списочных структур* (списков). Для организа-

ции списков используются свойства, начинающиеся строкой “refers_to_next”. Для представления модификатора сервисного примитива используется класс Service_primitive_modifier. Модификатор может иметь два значения (Истина/Ложь или +/–).

На рис. 7.10 приводится семантическая сеть для определения СИФБ. Для упрощения в приведенной диаграммы не представлены классы, относящиеся к адаптерам.

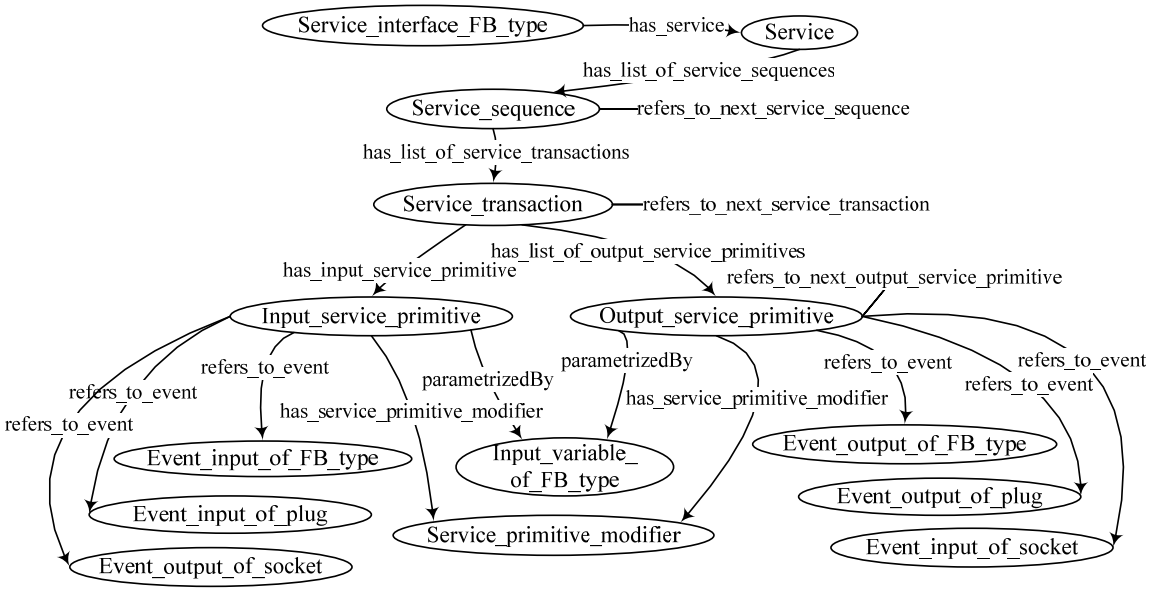


Рис. 7.10. Определение (типов) сервисных интерфейсных ФБ

7.1.9. Описание адаптеров

С адаптерами могут быть связаны четыре разные роли в зависимости от места применения адаптера: 1) адаптер-тип; 2) адаптер-экземпляр; 3) адаптер как агрегативный вход/выход у компонентных ФБ и субприложений; 4) адаптер как агрегативный вход/выход у ФБ-типов и субприложений-типов.

Первые три роли хорошо определены в стандарте и нашли свое отражение в онтологии. Адаптеры-типы представляются классом adapter_type (рис. 7.1), а адаптеры-экземпляры – классом component_adapter (рис. 7.2). Для представления адаптерных входов/выходов (роль 3) используется класс adapter_of_FB_like_instance. Таксономия классов, образуемая этим классом, приведена на рис. 7.11.

Следует заметить, что в сетях ФБ адаптерные входы/выходы обозначены знаком >>. В дальнейшем для простоты будем называть интерфейс ФБ-типов и субприложений-типов *оболочкой*. Адаптерный вход оболочки должен подаваться *как одно целое* на адаптер-

ный вход компонентного ФБ или субприложения и, соответственно, адаптерный выход компонентного ФБ или субприложения должен подаваться *как единое целое* на адаптерный выход оболочки (рис. 7.12). Несмотря на некоторую схожесть ролей 2 и 4, их следует различать. В отличие от роли 4 в роли 2 используется интерфейс адаптера-компонента. В роли 4 интерфейс адаптера вообще не рассматривается (во внимание принимается только имя адаптера). Адаптер может использоваться в двух ролях 2 и 4 одновременно, например, при определении типа составного ФБ (рис. 7.13).

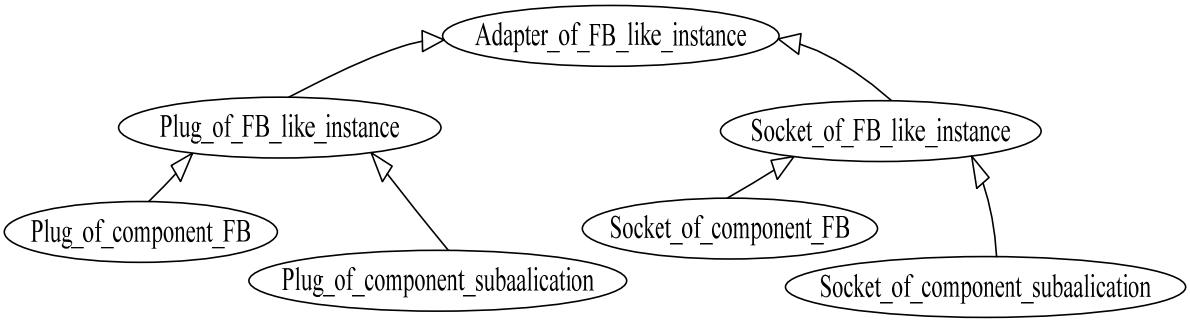


Рис. 7.11. Таксономия адаптерных входов-выходов

Следует отметить, что роль 4 адаптера в стандарте, по сути, не определена, примеров использования адаптеров в роли 4 также не нашлось, поэтому в онтологии ФБ данная роль не нашла своего отражения.

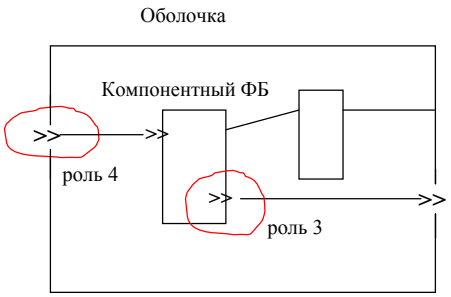


Рис. 7.12. Адаптерные входы и выходы ФБ-типов и субприложений-типов

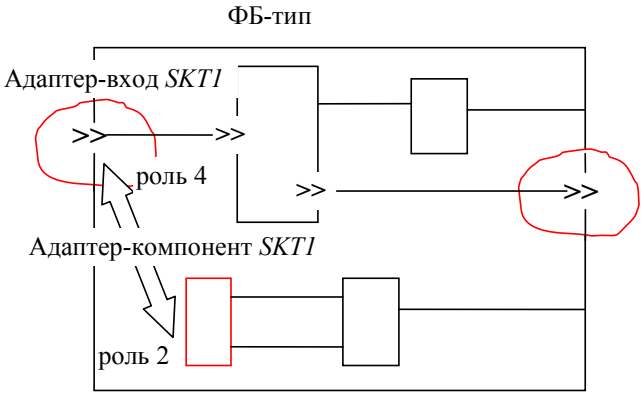


Рис. 7.13. Использование адаптера в двух ролях одновременно

7.1.10. Описание системных конфигураций

Для использования преимуществ модульного проектирования в

стандарте введена типизация не только на уровне ФБ, но и на уровне устройств и ресурсов. Отношение типизации определяет, по сути, отношение структурного наследования. Это означает, что сеть ФБ, определенная в типе, добавляется к сети ФБ, определенной в экземпляре. Рассмотрим проблему наследования и тиражирования экземпляров ресурсов и ФБ при использовании типов ресурсов и устройств на основе рис. 7.14.

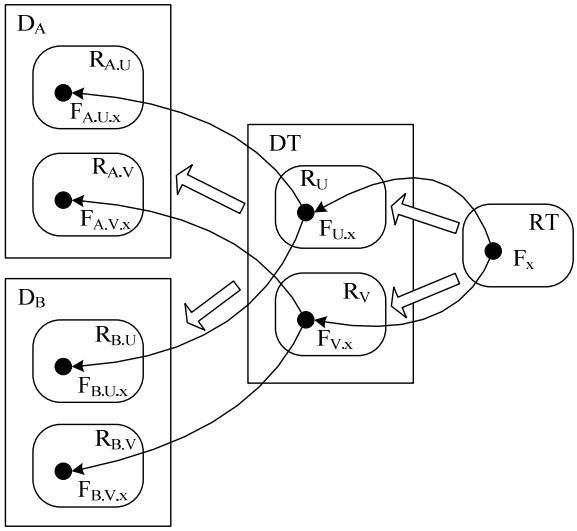


Рис. 7.14. Тиражирование ресурсов и ФБ в условиях типизации

На рис. 7.14 приняты следующие обозначения: буквой *D* обозначены устройства, буквой *R* – ресурсы, а буквой *F* – функциональные блоки (в определенной мере можно назвать их экземплярами). На месте индекса стоит имя элемента – устройства, ресурса или функционального блока. Если в имени встречается точка, то имя является составным и отражает иерархическую вложенность элементов. Через *DT* обозначен некоторый тип устройства, а через *RT* – некоторый тип ресурса. Широкие стрелки используются для обозначения связи между экземплярами ресурсов и устройств и соответствующими им типами. Стрелка идет от типа к экземпляру, показывая, таким образом, также и отношение подстановки содержимого типов в соответствующие экземпляры. Тонкие стрелки показывают связь компонентного ФБ в типе с соответствующим компонентным ФБ в экземпляре ресурса или устройства. Следует также заметить, что в ряде случаев нельзя однозначно считать некоторый элемент экземпляром вследствие относительности точек зрения. С одной точки зрения элемент будет экземпляром, а с другой – это некий тиражируемый подтип.

Тиражирование ресурсов и ФБ происходит при развертывании (или раскрытии) конфигурации. Суть процесса *развертывания* со-

стоит в замене ссылок на типы (устройств/ресурсов) на содержимое соответствующих типов.

Для онтологического описания отрицательным моментом является то, что до процесса развертывания мы не сможем определить реальные экземпляры ресурсов и ФБ, используемые в устройствах (и в системе). В отображении *mapping* используются не реальные экземпляры ФБ, а ссылки на них, представляющие иерархические имена (класс `FB_instance_reference` для ссылки на ФБ в приложении и класс `FB_resource_reference` для ссылки на ФБ на ресурсе). В то же время возможно установление прямой связи ссылок с соответствующими компонентными ФБ, входящими в состав ресурсов и являющимися «прообразами» данных ссылок. Примером такого прообраза на рис. 7.14 будет ФБ-экземпляр F_x.

Субонтология системных конфигураций представлена на рис. 7.15.

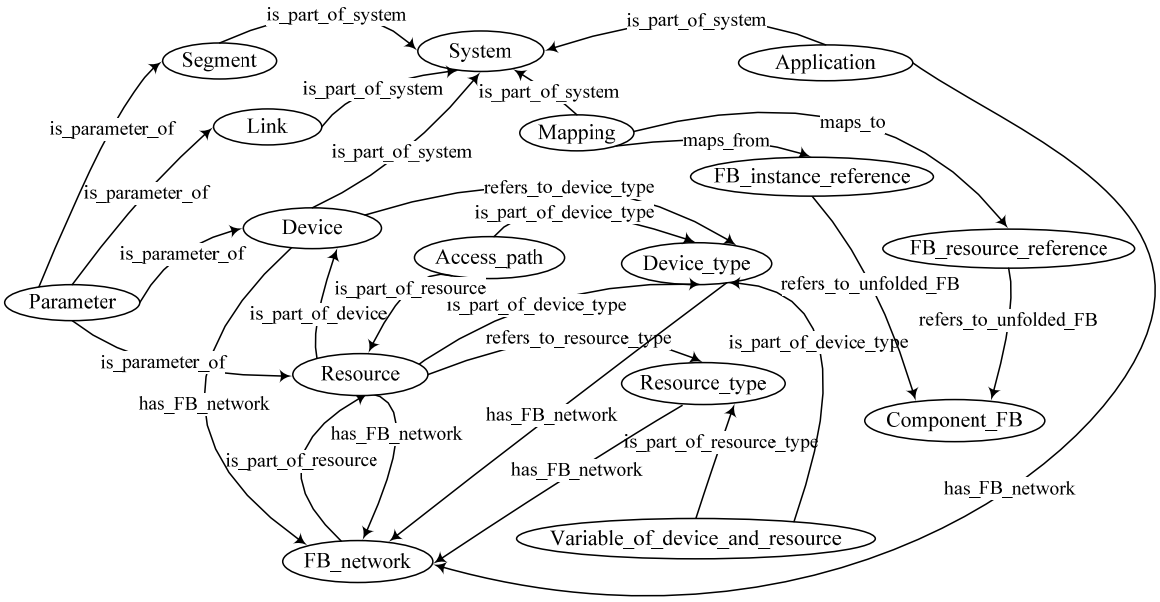


Рис. 7.15. Субонтология системных конфигураций

7.2. Свойства, связанные с семантической корректностью описаний

Семантический анализ описаний по своей сути сводится к определению семантически правильных конструкций в множестве синтаксически правильных. Синтаксическая правильность *XML*-описаний проектов стандарта IEC 61499 определяется соответствующим *DTD*-блоком [164], а ее проверка поддерживается существующими *XML*-парсерами. Ниже представлены возможности онтологии в «чисто» в семантическом описании и семантическом анали-

зе систем ФБ. Рассматриваемые семантические свойства никоим образом не затронуты в синтаксических описаниях.

Семантический анализ основан на определении так называемых «семантически правильных классов» и их вычислении в ходе логического вывода. *Семантически правильный класс* строится на основе соответствующего базового «синтаксически правильного» класса, но при этом в его определение включаются дополнительные семантические свойства и ограничения, полученные из анализа текста стандарта или практики его использования. Техника проведения семантического анализа основана на обычной процедуре классификации, осуществляемой штатной системой рассуждений (*reasoner*). Собственно семантический анализ сводится к сравнению множеств экземпляров, отнесенных как к базовому классу, так и к его семантически правильному аналогу. Равенство данных множеств свидетельствует о семантической корректности описания. Неравенство множеств свидетельствует о наличии ошибки в описании. Локализация ошибки сводится к поиску экземпляров в базовом классе, не отнесенных к семантически правильному классу.

В семантическом плане онтология позволяет:

1) находить «семантически правильные» событийные входы ФБ-типов (класс `Semantic_right_event_input_of_FB_type`);

2) находить «семантически правильные» событийные входы компонентных ФБ (класс `Semantic_right_event_input_of_component_FB`);

3) находить «семантически правильные» событийные входы субприложений-типов (класс `Semantic_right_event_input_of_subapplication_type`);

4) находить «семантически правильные» событийные выходы ФБ-типов (класс `Semantic_right_event_output_of_FB_type`);

5) находить «семантически правильные» событийные выходы компонентных ФБ (класс `Semantic_right_event_output_of_component_FB`);

6) находить «семантически правильные» событийные выходы субприложений-типов (класс `Semantic_right_event_output_of_subapplication_type`);

7) находить «семантически правильные» входные переменные ФБ-типов (класс `Semantic_right_input_variable_of_FB_type`);

8) находить «семантически правильные» входные переменные

компонентных ФБ (класс Semantic_right_input_variable_of_component_FB);

9) находить «семантически правильные» выходные переменные ФБ-типов (класс Semantic_right_output_variable_of_FB_type);

10) находить «семантически правильные» выходные переменные компонентных ФБ (класс Semantic_right_output_ variable_of_component_FB);

11) находить «семантически правильные» событийные и информационные связи в приложении (класс Right_appl_event_connection).

Ниже приводятся определения семантически правильных концептов, относящихся к ФБ, а также определения соответствующих «семантически правильных» классов (на основе формализма дескриптивной логики – *DL*).

Определение 7.1. Событийный вход ФБ-типа является семантически правильным, если верно одно из двух утверждений: 1) он является частью интерфейса базисного ФБ-типа и при этом используется в некотором условии *EC*-перехода этого ФБ-типа; 2) он является частью интерфейса составного ФБ-типа и при этом из него выходит некоторая (внутренняя) событийная связь.

Выражение *DL*-логики:

```
Semantic_right_event_input_of_FB_type ≡
Event_input_of_FB_type ⊓
(∃ is_checked_by.EC_transition
⊓ =1
is_part_of_interface.Interface_of_basic_FB_type)
⊔ (∃ origi-
nates_event_connection.Event_connection
⊓ =1
is_part_of_interface.Interface_of_composite_FB_ty
pe)
```

Определение 7.2. Событийный вход компонентного ФБ является семантически правильным, если в него входит, по крайней мере, одна событийная дуга (т.е. этот вход не является «висячим»).

Выражение *DL*-логики:

```
Semantic_right_event_input_of_component_FB ≡
```

Event_input_of_component_FB ⊓ ∃ finish-
es_event_connection.Event_connection

Определение 7.3. Событийный вход субприложения-типа является семантически правильным, если из него выходит некоторая (внутренняя) событийная связь (т.е. этот вход задействован).

Выражение *DL*-логики:

Seman-
tic_right_event_input_of_subapplication_type ≡
Event_input_of_subapplication_type
⊓ ∃ origi-
nates_event_connection.Event_connection

Определение 7.4. Событийный выход ФБ-типа является семантически правильным, если верно одно из двух утверждений: 1) он является частью интерфейса составного ФБ-типа и при этом в нем оканчивается по меньшей мере одна событийная дуга; 2) он является частью интерфейса базисного ФБ-типа и при этом он используется по меньшей мере в одной *EC*-акции, входящей в состав *ECC* ФБ-типа.

Выражение *DL*-логики:

Semantic_right_event_output_of_FB_type ≡
Event_output_of_FB_type ⊓ (∃ finish-
es_event_connection.Event_connection
⊓ =1
is_part_of_interface.Interface_of_composite_FB_ty
pe)
⊔ (∃ is_issued_by.EC_action
⊓ =1
is_part_of_interface.Interface_of_basic_FB_type)

Определение 7.5. Событийный выход компонентного ФБ является семантически правильным, если из него выходит, по крайней мере, одна событийная связь (т.е. этот выход не является «висячим»).

Выражение *DL*-логики:

Semantic_right_event_output_of_component_FB ≡
Event_output_of_component_FB
⊓ ∃ origi-
nates_event_connection.Event_connection

Определение 7.6. Событийный выход субприложения-типа яв-

ляется семантически правильным, если в него входит некоторая (внутренняя) событийная связь (т.е. этот выход задействован).

Выражение *DL*-логики:

```
    Seman-
tic_right_event_output_of_subapplication_type ≡
    Event_output_of_subapplication_type
    ⊓ ∃ finishes_event_connection.Event_connection
```

Определение 7.7. Входная переменная ФБ-типа является семантически правильной, если она связана *WITH*-связью по крайней мере с одним событийным входом этого ФБ-типа и верно одно из двух утверждений: 1) если это базисный ФБ-тип, то эта входная переменная используется при вычислениях в каких-либо алгоритмах или сторожевых условиях этого ФБ-типа; 2) если это составной ФБ-тип, то из этой входной переменной выходит по крайней мере, одна (внутренняя) информационная связь. Таким образом, при выполнении вышеперечисленных условий данная входная переменная будет задействована.

Выражение *DL*-логики:

```
    Semantic_right_input_variable_of_FB_type ≡
    Input_variable_of_FB_type ⊓
(∃ is_used_in.(Algorithm ⊔ Guard_condition)
    ⊓ =1
is_part_of_interface.Interface_of_basic_FB_type)
    ⊔ (∃ origi-
nates_data_connection.Data_connection
    ⊓ =1
is_part_of_interface.Interface_of_composite_FB_ty
pe)
    ⊓
∃ is_linked_with_event_input.
    Event_input_of_FB_type
```

Определение 7.8. Входная переменная компонентного ФБ является семантически правильной, если в нее входит по крайней мере одна информационная связь (т.е. этот вход не является «висячим»).

Выражение *DL*-логики:

```
    Semantic_right_input_variable_of_component_FB
≡
```

Input_variable_of_component_FB
⊓ =1 finishes_data_connection.Data_connection

Определение 7.9. Выходная переменная ФБ-типа является семантически правильной, если она связана *WITH*-связью по крайней мере с одним событийным выходом этого ФБ-типа и верно одно из двух утверждений: 1) если это базисный ФБ-тип, то эта выходная переменная должна модифицироваться по крайней мере одним алгоритмом, входящим в состав этого ФБ-типа; 2) если это составной ФБ-тип, то в эту выходную переменную должна входить ровно одна информационная связь.

Выражение *DL*-логики:

Semantic_right_output_variable_of_FB_type ≡
output_variable_of_FB_type ⊓
(∃ is_modified_by.Algorithm
⊓ =1
is_part_of_interface.Interface_of_basic_FB_type)
⊔ (=1 finishes_data_connection.Data_connection
⊓ =1
is_part_of_interface.Interface_of_composite_FB_type)
⊓
∃ is_linked_with_event_output.
Event_output_of_FB_type

Определение 7.10. Выходная переменная компонентного ФБ является семантически правильной, если из нее выходит по крайней мере одна информационная связь (т.е. этот выход не является «висячим»).

Выражение *DL*-логики:

Seman-
tic_right_output_variable_of_component_FB ≡
Output_variable_of_component_FB
⊓ ∃ originates_data_connection.Data_connection

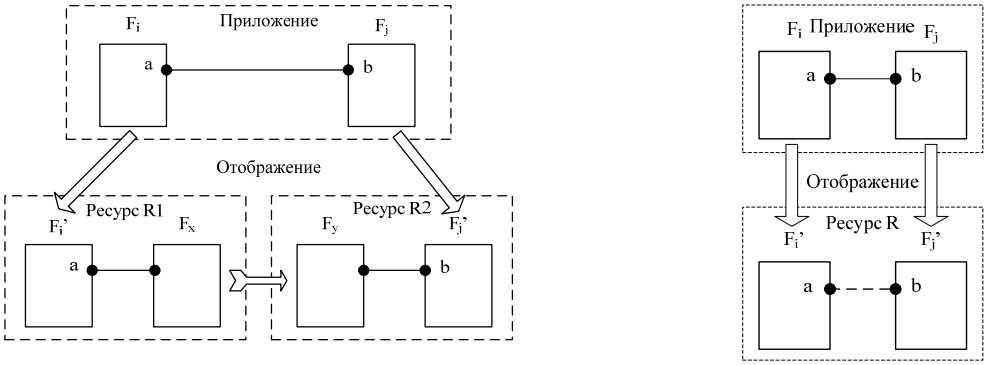
Определение 7.11. Связь (событийная или информационная) вида $F_{i.}a \rightarrow F_{j.}b$ между двумя ФБ в приложении является семантически правильной, если верны следующие два утверждения:

1) если блоки F_i и F_j отображаются на разные ресурсы (пусть в виде блоков $F_i^{'}$ и $F_j^{'}$ соответственно), то из выхода с именем *a* блока $F_i^{'}$ должна исходить связь к некоторому коммуникационному

СИФБ, а к входу с именем b блока $F_j^{'}$ должна подходить связь из некоторого коммуникационного СИФБ;

2) если блоки F_i и F_j отображаются на один и тот же ресурс (пусть в виде блоков $F_i^{'}$ и $F_j^{'}$ соответственно), то между блоками $F_i^{'}$ и $F_j^{'}$ должна существовать связь вида $F_{i.}^{'}a \rightarrow F_{j.}^{'}b.$

Иллюстрация существа данных правил приведена на рис. 7.16.



а) *б*)

Рис. 7.16. Распределение функциональных блоков приложения:
а – на разные ресурсы; *б* – на один и тот же ресурс

Следует заметить, что нельзя разработать соответствующий класс `Right_appl_event_connection` для описания правильной связи в приложении только с помощью логики *DL*. Ее описательных возможностей здесь явно недостаточно. Для описания соответствующих необходимых подклассов используются правила языка *SWRL*. В этом случае описание правильной событийной связи в приложении выражается следующим образом:

```
Right_appl_event_connection ≡
Right_appl_event_connection1 ⊔
Right_appl_event_connection2
```

Входящие в состав этой формулы классы `Right_appl_event_connection1` и `Right_appl_event_connection2` описываются ниже с использованием *SWRL*-правил (рис. 7.17 и 7.18).

```
Event_connection(?c1) ∧ comes_from_event(?c1,?o1)
∧ comes_to_event(?c1,?i1) ∧ Application(?a1) ∧
has_FB_network(?a1,?w1) ∧
is_part_of_FB_network(?c1,?w1) ∧
is_part_of_interface(?o1,?x1) ∧
```

```
has_interface(?b1,?x1) ∧
is_part_of_FB_network(?b1,?w1) ∧ has_name(?o1,?n1) ∧
refers_to_FB_type(?b1,?t1) ∧ Mapping(?m1) ∧
maps_from(?m1,?u1) ∧ maps_to(?m1,?u3) ∧
refers_to_unfolded_FB(?u1,?b1) ∧
refers_to_unfolded_FB(?u3,?b3) ∧
refers_to_FB_type(?b3, ?t1) ∧ has_interface(?b3,?x3)
∧ is_part_of_FB_network(?b3,?w2) ∧ Resource(?r1) ∧
has_FB_network(?r1,?w2) ∧
is_part_of_FB_network(?c2,?w2) ∧
is_part_of_FB_network(?b5,?w2) ∧
Event_connection(?c2) ∧ comes_from_event(?c2,?o2) ∧
comes_to_event(?c2,?i3) ∧
is_part_of_interface(?i3,?x5) ∧
has_interface(?b5,?x5) ∧
is_part_of_interface(?o2,?x3) ∧ has_name(?o2,?n1) ∧
Service_interface_FB_type(?t3) ∧
refers_to_FB_type(?b5,?t3) ∧
is_part_of_interface(?i1,?x2) ∧
has_interface(?b2,?x2) ∧
is_part_of_FB_network(?b2,?w1) ∧ has_name(?i1,?n2) ∧
refers_to_FB_type(?b2,?t2) ∧ Mapping(?m2) ∧
maps_from(?m2,?u2) ∧ maps_to(?m2,?u4) ∧
refers_to_unfolded_FB(?u2,?b2) ∧ re-
fers_to_unfolded_FB(?u4,?b4) ∧ re-
fers_to_FB_type(?b4, ?t2) ∧ has_interface(?b4,?x4) ∧
is_part_of_FB_network(?b4,?w3) ∧ Resource(?r2) ∧
has_FB_network(?r2,?w3) ∧
is_part_of_FB_network(?c3,?w3) ∧
is_part_of_FB_network(?b6,?w3) ∧
Event_connection(?c3) ∧ comes_from_event(?c3,?o3) ∧
comes_to_event(?c3,?i2) ∧
is_part_of_interface(?o3,?x6) ∧
has_interface(?b6,?x6) ∧
is_part_of_interface(?i2,?x4) ∧ has_name(?i2,?n2) ∧
Service_interface_FB_type(?t4) ∧
refers_to_FB_type(?b6,?t4) ∧
```

```
DifferentFrom(?b1,?b2) ∧ DifferentFrom(?r1,?r2) →
Right_appl_event_connection1(?c1)
```

Рис. 7.17. Правило *SWRL* для вычисления правильных событийных связей в приложении (случай распределения ФБ на разные ресурсы)

```
 Event_connection(?c1) ∧ comes_from_event(?c1,?o1)
∧ comes_to_event(?c1,?i1) ∧ Application(?a1) ∧
has_FB_network(?a1,?w1) ∧
is_part_of_FB_network(?c1,?w1) ∧
is_part_of_interface(?o1,?x1) ∧
has_interface(?b1,?x1) ∧
is_part_of_FB_network(?b1,?w1) ∧
refers_to_FB_type(?b1,?t1) ∧ Mapping(?m1) ∧
maps_from(?m1,?u1) ∧ maps_to(?m1,?u3) ∧
refers_to_unfolded_FB(?u1,?b1) ∧
refers_to_unfolded_FB(?u3,?b3) ∧
refers_to_FB_type(?b3, ?t1) ∧
is_part_of_FB_network(?b3,?w2) ∧ Resource(?r) ∧
has_FB_network(?r,?w2) ∧
is_part_of_interface(?i1,?x2) ∧
has_interface(?b2,?x2) ∧
is_part_of_FB_network(?b2,?w1) ∧ re-
fers_to_FB_type(?b2,?t2) ∧ Mapping(?m2) ∧
maps_FB_from(?m2,?b2) ∧ maps_FB_to(?m2,?b4) ∧
refers_to_unfolded_FB(?u2,?b2) ∧
refers_to_unfolded_FB(?u4,?b4) ∧
refers_to_FB_type(?b4, ?t2) ∧
is_part_of_FB_network(?b4,?w2) →
Right_appl_event_connection2(?c1)
```

Рис. 7.18. Правило *SWRL* для вычисления правильных событийных связей в приложении (случай распределения ФБ на один и тот же ресурс)

Для более полного визуального восприятия данных правил, а также их визуальной проверки могут быть составлены соответствующие графы запросов [68]. Граф запроса наглядно описывает си-

туацию в виде некоторой семантической сети, узлами которой являются переменные. Эти переменные так же, как и в правиле *SWRL*, будем предварять знаком вопроса (?). Дугами в графе запроса являются отношения между соответствующими классами, представляющими интерес. Графы запросов для ситуаций, представленных на рис. 7.17 и 7.18, приведены на рис. 7.19 и 7.20 соответственно. Ограничениями по переменным в этом случае являются *?b*1≠*?b2* и *?r*1≠*?r*2.

Следует отметить, что для повышения эффективности вычислений и упрощения отладки (следуя модульному принципу) приведенные правила можно было бы разбить на несколько правил меньшей размерности. Однако в данном случае разбиение правила на модули для иллюстративных целей большого смысла не имеет.

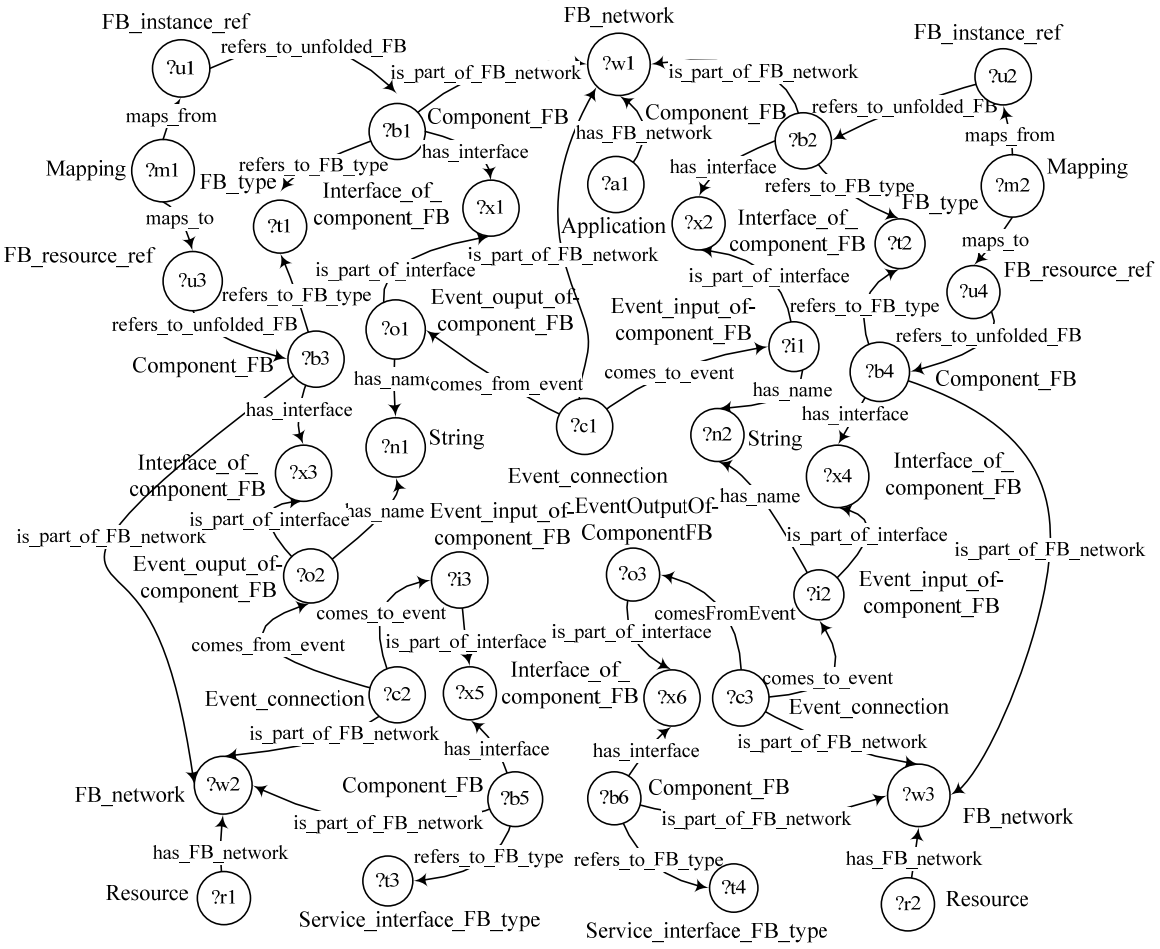


Рис. 7.19. Граф запроса для ситуации
о распределении ФБ на разные ресурсы

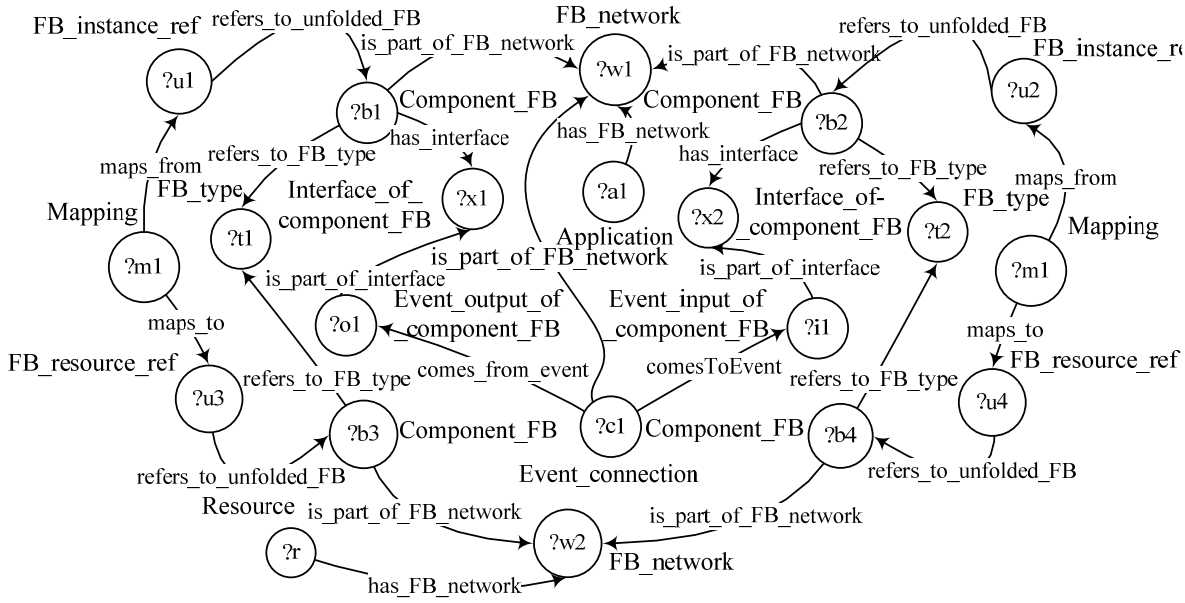


Рис. 7.20. Граф запроса для ситуации о распределении ФБ на один и тот же ресурс

7.3. Обнаружение циклов с использованием графов зависимостей

Метод семантического анализа проектов IEC 61499 на основе онтологий может быть дополнен обнаружением циклов в системах ФБ. Наличие циклов, не содержащих блоки временной задержки *E_DELAY*, при использовании определенных моделей выполнения (например, *NPMTR* [219]) может привести к зацикливанию системы, что является серьезной ошибкой. Ручное выявление циклов, особенно в многоуровневых иерархических структурах, является нетривиальной задачей.

Ниже рассматривается задача детектирования циклов в иерархических системах ФБ. Задача решалась в рамках онтологического моделирования систем ФБ с использованием *OWL DL* (типа *SROIQ*) [224, 250] и *SWRL* [221] в инструментальной системе *Protégé* [201]. При разработке новых онтологий для определения циклов использовался терминологический словарь онтологии, предложенной в предыдущих разделах.

7.3.1. Постановка задачи и анализ

Сети ФБ являются распространенным артефактом проектирования систем ФБ. Они как составная часть включаются в составные ФБ (типы), ресурсы и типы ресурсов, устройства и типы устройств. Фактически сети ФБ представляют собой плоскую графовую струк-

туру, основными узлами которой являются компонентные ФБ. Назовем подобную структуру событийным графом. Он является одной из разновидностей графов зависимостей [54].

В общем случае событийный граф в «плоской» системе ФБ определяется двойкой $G = (FB, E)$, где FB – конечное множество (экземпляров) ФБ; $E \subseteq FB \times FB$ – множество событийных дуг. Считается, что существует дуга $(fb_i, fb_j) \in E$, если из блока fb_i исходит хотя бы одна событийная связь в блок fb_j. В иерархических системах ФБ определение событийного графа несколько сложнее, но может быть просто выведено из формальной модели систем ФБ [117].

Поиск циклов в сетях ФБ осложняется неопределенностями следующего плана: точно неизвестно, является ли ФБ, входящий в цикл, разрывом в прохождении сигнала или нет. Наиболее неопределенным является составной ФБ поскольку его внутренность на уровне плоской структуры неизвестна. Более определенным является базисный ФБ, поскольку его структура известна, но точно прохождение через него сигнала можно определить только исходя из контекста его выполнения, что на стадии семантического (статического) анализа невозможно. Исходя из степени неопределенности в прохождении сигналов через ФБ можно различать и степени неопределенности опасных циклов. Для простоты далее будем считать, что базисные ФБ являются «прозрачными» для прохождения сигнала. Цикл в сети ФБ будем называть *опасным*, если он состоит только из блоков, отличных от блока *E_DELAY*.

Поскольку в общем случае системы ФБ являются иерархическими, то при определении циклов следует учитывать и межуровневые связи. При этом вследствие увеличения степени определенности может существенно сократиться объем выдаваемой неактуальной и «путающей» информации, относящейся к циклам. На рис. 7.21 на примере представлена взаимосвязь соседних уровней иерархии системы ФБ – верхнего и нижнего. В данном случае на каждом из уровней размещено по одной сети ФБ. Составной ФБ *fb*1 (типа *t*1) сети верхнего уровня «развертывается» в соответствующую сеть нижнего уровня. Структура данной сети изоморфна структуре типа *t*1, включая и интерфейс. В дальнейшем будем называть такие сети «клонами типа» или просто «клонами».

На рис. 7.21 входу *e*1 блока *fb*1 соответствует одноименный вход *e*2 соответствующего клона типа, а выходу *e*6 – выход *e*5. Сигнал, пришедший на вход *e*1 блока *fb*1, передается на вход *e*2 клона типа, а затем на вход *e*3 блока *fb*2 нижнего уровня.

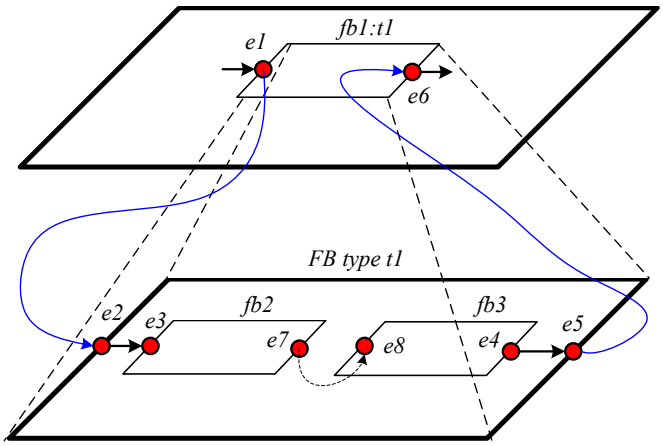


Рис. 7.21. Взаимосвязь соседних уровней иерархии в системе ФБ

Очевидно, что передача сигналов с одного уровня на другой определяет некоторые отношения между ФБ разных уровней иерархии. Будем говорить, что между ФБ fb_i и fb_j существует *нисходящая межуровневая связь*, если некоторому входу блока fb_i соответствует одноименный вход в клоне типа (соответствующего блоку fb_i), который в свою очередь связан с некоторым входом блока fb_j. Аналогично между компонентными ФБ можно определить *восходящую межуровневую связь*. Подразумевается, что все упомянутые выше связи являются событийными, хотя при необходимости то же самое можно определить и в отношении связей по данным. Из рис. 7.21 можно видеть, что связь (*fb*1, *fb*2) является нисходящей, а связь (*fb*3, *fb*1) – восходящей межуровневой связью. Также можно заметить, что в системе, представленной на рис. 7.21, будет цикл, если существует цепь передачи сигнала от выхода *e*7 блока *fb*2 к входу *e*8 блока *fb*3 на нижнем уровне, а также цепь от выхода *e*6 к входу *e*1 блока *fb*1 на верхнем уровне.

Исходная система ФБ («как есть») представляется в *неразвернутом* виде как некоторый набор типов (ФБ, ресурсов, устройств). Ядром для развертки системы может служить, например, системная конфигурация (файл **.sys*) или даже некоторый тип ФБ. Развертка системы заключается в развертке всех составных ФБ, ресурсов и устройств. Развертка составного ФБ сводится к добавлению в системную иерархию клона своего типа на уровень ниже, чем включающая сеть этого составного ФБ. Как можно понять, развертывание системы порождает новые экземпляры (ФБ и ресурсов). Однако в онтологической модели нельзя создавать новые экземпляры (иначе, *individuals*) средствами самой этой модели, поэтому проведение развертывания системы в онтологической модели в принципе *не-*

возможно. Конечно, это можно сделать внешними средствами, как это предлагалось ранее при анализе корректности связей в приложении при распределении на разные ресурсы.

Тем не менее использование фактора иерархичности при обнаружении циклов все же возможно даже в неразвернутой системе. Для решения этой задачи воспользуемся искусственным приемом – примем, что в системе существует только по одному экземпляру клона каждого типа. Пусть, например, в системе существует N экземпляров ФБ типа X. В этом случае все N клонов типа X, соответствующие экземплярам, «стягиваются» в один абстрактный клон типа X. Следует заметить, что подобная «стянутая» система есть не что иное, как неразвернутая система ФБ, в которой сам тип ФБ является абстрактным клоном этого типа.

Естественно, что при подобном «стягивании» в системе могут появиться неправильные связи и несуществующие в системе циклы. Иллюстрация данного утверждения приведена на рис. 7.22.

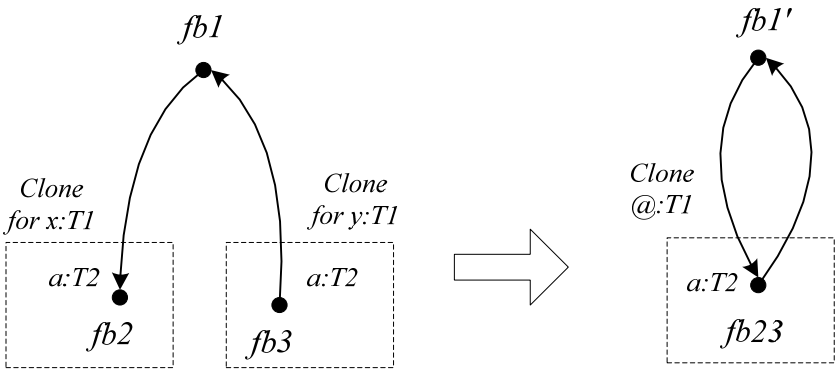


Рис. 7.22. Пример «стягивания» системы
с образованием не специфицированного цикла

На данном рисунке приведен пример «стягивания» двух клонов типа $T1$, соответствующих экземплярам ФБ x и y, в один абстрактный клон типа $T1$ (обозначен символом @)). Вершины приведенных графов представляют экземпляры ФБ, а стрелки – межуровневые связи. Имена *fb*1, *fb*2, *fb*23, *fb*1' соответствуют уникальным идентификаторам экземпляров ФБ в системе. Имя компонентного ФБ записывается в виде пары в формате *Имя_ФБ:Имя_типа*, например, *a:T*2. Следует отметить, что родителем экземпляров ФБ *fb*2, *fb*3 и *fb*23 является один и тот же компонентный ФБ *a:T*2. На рис. 7.22 слева показана развернутая система ФБ, а справа – ее «стянутый» вариант. Как видно из данного рисунка, в результирующей системе появился не специфицированный цикл.

В то же время можно утверждать, что циклы исходной (развернутой) системы сохраняются в результирующей («стянутой») системе. Однако при этом идентификация узлов, входящих в цикл, будет определяться не в виде уникальных идентификаторов экземпляров ФБ (как в случае развернутой системы), а в виде пары (*Имя типа ФБ*, *Имя компонентного ФБ в типе*). При этом определенная информация будет, конечно, потеряна.

При анализе неразвернутой системы ФБ следует руководствоваться следующим правилом: «Если в неразвернутой системе ФБ не обнаружены циклы, то их также нет в развернутой системе». Наличие в неразвернутой системе ФБ циклов – это *необходимое*, но не *достаточное* условие существования циклов в развернутой системе. Следует отметить, что если в системе ФБ используется уникальная типизация ФБ или если типизация не уникальна только на последнем уровне иерархии, то предложенный выше метод будет также работать в неразвернутой системе правильно без всяких оговорок.

7.3.2. Разработка правил *SWRL*

Ниже последовательно, методом «снизу-вверх» приводится решение задачи определения опасных циклов в иерархических системах ФБ.

Свойство `OBJ_event_connect_to_OBJ`, определяющее взаимосвязь объектов в системе на основе событийных связей, может быть выражено двумя путями: а) с помощью дескриптивной логики; б) с помощью *SWRL*-правил.

В первом случае данное свойство определяется как цепочка свойств. Иначе эту цепочку можно назвать композицией отношений:

$$\texttt{OBJ\_event\_connect\_to\_OBJ} \sqsubseteq \texttt{has\_interface} \circ \texttt{is\_part\_of\_interface}^{-} \circ \texttt{comes\_from\_event}^{-} \circ \texttt{comes\_to\_event} \circ \texttt{is\_part\_of\_interface} \circ \texttt{has\_interface}^{-}$$

Здесь символ ° обозначает операцию композиции. Инверсное свойство помечается знаком *минус* на месте верхнего индекса. В системе *Protégé* это свойство выражается следующим образом:

```
has_interface o inverse
(is_part_of_interface) o inverse
(comes_from_event) o comes_to_event o
is_part_of_interface o inverse (has_interface)
```

Во втором случае свойство `OBJ_event_connect_to_OBJ` определяется в виде следующего правила:

```
   has_interface(?x,?i1),
is_part_of_interface(?e1,?i1),
comes_from_event(?c,?e1), comes_to_event(?c,?e2),
is_part_of_interface(?e2,?i2),
has_interface(?y,?i2) ->
OBJ_event_connect_to_OBJ(?x,?y)
```

Правило для определения нисходящей межуровневой связи представлено на рис. 7.23.

```
Component_FB(?fb1), Composite_FB_type(?t), re-
fers_to_FB_type(?fb1,?t), has_interface(?fb1,?i1),
event_input_of_FB_like_instance(?e1),
is_part_of_interface(?e1,?i1), has_name(?e1,?n),
has_interface(?t,?i2),
event_input_of_FB_like_type(?e2),
is_part_of_interface(?e2,?i2), has_name(?e2,?n),
has_FB_network(?t,?fnet),
is_part_of_FB_network(?fb2,?fnet),
has_interface(?fb2,?i3),
event_input_of_FB_like_instance(?e3),
is_part_of_interface(?e3,?i3), event_connection(?c),
comes_from_event(?c,?e2), comes_to_event(?c,?e3) ->
down_interlevel_fb_conn_to(?fb1,?fb2)
```

Рис. 7.23. Правило для определения нисходящей межуровневой связи

Графическое представление данного правила в виде графа запросов приведено на рис. 7.24. *Пунктирной* стрелкой определяется *результирующее* свойство.

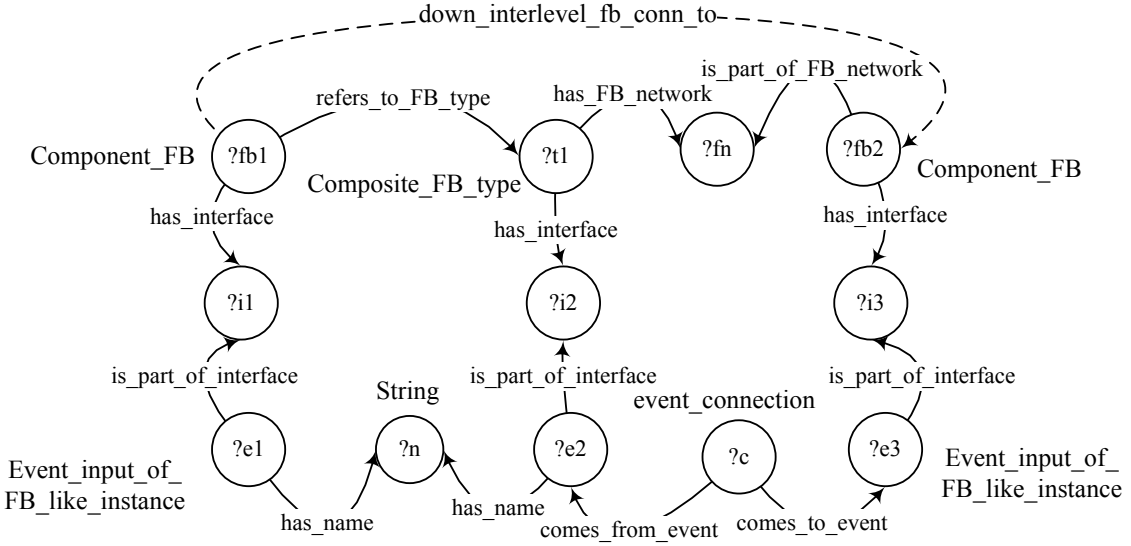


Рис. 7.24. Графическое представление правила для определения нисходящей межуровневой связи

Похожим образом определяется правило для восходящей межуровневой связи (рис. 7.25).

```
Component_FB(?fb1), Composite_FB_type(?t),
refers_to_FB_type(?fb1,?t), has_interface(?fb1,?i1),
Event_output_of_FB_like_instance(?e1),
is_part_of_interface(?e1,?i1), has_name(?e1,?n),
has_interface(?t,?i2),
Event_output_of_FB_like_type(?e2),
is_part_of_interface(?e2,?i2), has_name(?e2,?n),
has_FB_network(?t,?fnet),
is_part_of_FB_network(?fb2,?fnet),
has_interface(?fb2,?i3),
event_output_of_FB_like_instance(?e3),
is_part_of_interface(?e3,?i3),event_connection(?c),
comes_from_event(?c,?e3), comes_to_event(?c,?e2) ->
up_interlevel_fb_conn_to(?fb2,?fb1)
```

Рис. 7.25. Правило для определения восходящей межуровневой связи

В нашей онтологической модели иерархической системы ФБ (с полностью определенным набором типов) роль экземпляров составных ФБ по передаче сигналов перекладывается на межуровневые связи. Поэтому опасными будут те циклы, которые включают только экземпляры базисных ФБ, а также СИФБ, отличные от ФБ *E_DELAY* (как видно, экземпляры составных ФБ здесь не учитываются). В дальнейшем для простоты будем учитывать только базисные ФБ.

Связями, входящими в опасные циклы, являются связи между базисными ФБ, а также межуровневые связи. Определим соответствующее свойство `BFB_event_connect_to_BFB_in_ hierarchy` с помощью следующих правил (рис. 7.26):

```
Basic_FB(?x), Basic_FB(?y),
OBJ_event_connect_to_OBJ_rule(?x, ?y) ->
BFB_event_connect_to_BFB_in_hierarchy(?x, ?y)

down_interlevel_fb_conn_to(?x, ?y) ->
BFB_event_connect_to_BFB_in_hierarchy(?x, ?y)

up_interlevel_fb_conn_to(?x, ?y) ->
BFB_event_connect_to_BFB_in_hierarchy(?x, ?y)
```

Рис. 7.26. Правила для определения связей, входящих в опасные циклы

Найдем транзитивное замыкание отношения

BFB_event_connect_to_BFB_in_hierarchy_transitive =

$\text{BFB\_event\_connect\_to\_BFB\_in\_hierarchy}^{+}$

с помощью правил, приведенных на рис. 7.27.

```
BFB_event_connect_to_BFB_in_hierarchy(?x, ?y) ->
BFB_event_connect_to_BFB_in_hierarchy_transitive(?x,
?y)

BFB_event_connect_to_BFB_in_hierarchy(?x, ?y),
BFB_event_connect_to_BFB_in_hierarchy_transitive(?y,
?z) ->
BFB_event_connect_to_BFB_in_hierarchy_transitive(?x,
?z)
```

Рис. 7.27. Правила для вычисления транзитивного замыкания свойства BFB_event_connect_to_BFB_in_hierarchy

И, наконец, на основе введенных правил можно дать определение опасного цикла.

Определение 7.12. В иерархической (неразвернутой) системе ФБ существует опасный цикл, если в данной системе существует экземпляр базисного ФБ, в котором существует петля из дуги типа BFB_event_connect_to_BFB_in_hierarchy_transitive. Иными словами, в этом случае экземпляр ФБ достижим из самого себя.

Данное определение может быть использовано для вычисления класса Wrong_cycle_explicitly_in_FB_hierarchy – класса всех экземпляров базисных ФБ, входящих в опасные циклы (в неразвернутой структуре). Соответствующее правило приведено ниже:

```
basic_FB(?x),
BFB_event_connect_to_BFB_in_hierarchy_transitive(?x,
?x) -> Wrong_cycle_explicitly_in_FB_hierarchy(?x)
```

Следует отметить, что данное правило не предназначено для перечисления всех конкретных опасных циклов, но может быть полезно при ручном их определении.

7.3.3. Уточнение метода: учет диаграммы *ECC* базисных ФБ

Предположение о «прозрачности» базисных ФБ для прохождения сигналов, принятое в начале раздела, является условным. Ясно, что не каждое событие, поступающее на вход базисного ФБ, инициирует появление некоторого события на выходе этого блока. Для того, чтобы результаты поиска опасных циклов сделать еще более точными, следует учитывать диаграмму *ECC* в базисных ФБ и использовать соединения в циклах не на уровне ФБ, а на уровне событийных входов и выходов.

Утверждение. Событие на входе базисного ФБ может вызвать появление события на его выходе только тогда, когда это входное событие фигурирует в каком-нибудь *EC*-переходе диаграммы *ECC* этого блока и этот *EC*-переход связан (возможно, через цепочку *EC*-переходов, не нагруженных событиями) с таким *EC*-состоянием, с которым ассоциирована некоторая *EC*-акция, в состав которой входит событийный выход этого блока.

Данное утверждение является очевидным и в доказательстве не нуждается. Следует иметь в виду, что утверждение определяет необходимое, но не достаточное условие передачи события с входа на выход базисного ФБ. Для решения уточненной задачи поиска опасных циклов следует ввести понятие межуровневой связи на событийном уровне (например, с помощью свойств `down_interlevel_ev_conn_to` и `up_interlevel_ev_conn_to`), а также свойство `ev_through_ECC_to_ev`, отражающее передачу сигнала с входа на выход в базисном ФБ (на основе утверждения).

Для экономии места с помощью следующего *SWRL*-правила определим только второе свойство:

```
    Event(?e1), EC_transition(?t),
checks_event(?t,?e1), comes_to_EC_state(?t,?s2),
EC_action(?a), is_attached_to(?a,?s2), is-
sues_event(?a,?e2) -> ev_through_ECC_to_ev(?e1,?e2)
```

Здесь для простоты предполагается, что в диаграмме *ECC* существуют только *EC*-переходы, нагруженные событиями. Однако никаких принципиальных ограничений нет и для общего случая, однако при этом реализация усложняется.

7.3.4. Оценка метода на основе примера

Рассмотрим для примера иерархическую систему ФБ одного из дистрибутивов инструментального средства *FBDK* [134]. Корневым модулем данной системы является составной ФБ *XBAR_MVC*.

Иерархия данной системы, представленная на языке *UML-FB* [245], изображена на рис. 7.28. Как видно из данного рисунка, типизация в приведенной системе ФБ уникальна, кроме последнего уровня. Следовательно, полностью подходит описанный выше метод для поиска циклов в неразвернутых структурах. Было разработано онтологическое описание иерархической системы ФБ *XBAR_MVC*, которое в дальнейшем было дополнено правилами для поиска опасных циклов. С помощью ризонера *HermiT* 1.3.3 в системе *Protégé* была произведена классификация элементов системы из рис. 7.29. Вычисленный класс `Wrong_cycle_explicitly_in_FB_hierarchy = {ADF_FF, DRV, V_SEL, SEL2, SENSOR}` выделен на скриншоте системы *Protégé* контуром (см. рис. 7.29). Ручная проверка показала правильность автоматических вычислений, хотя первоначально вхождение некоторых ФБ в опасные циклы было не очевидным.

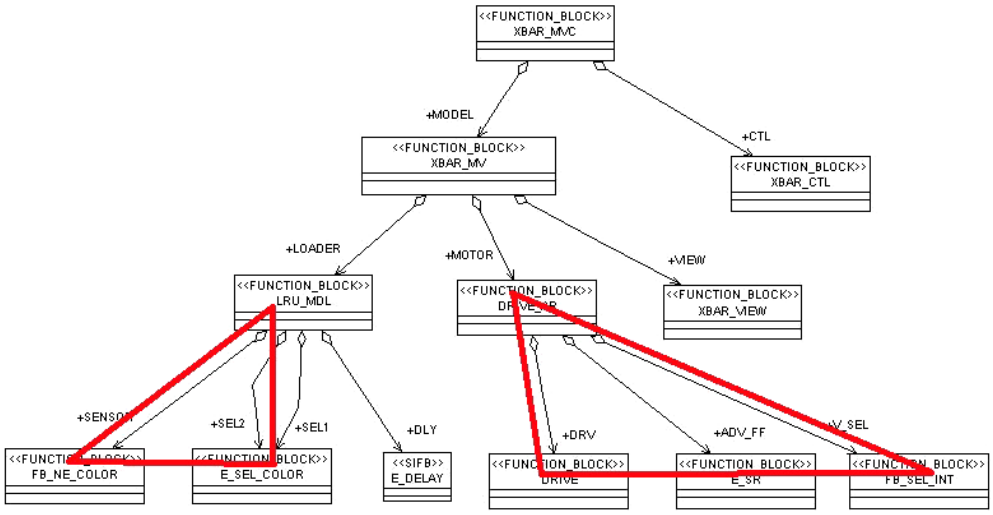


Рис. 7.28. Пример иерархической системы ФБ *XBAR_MVC*

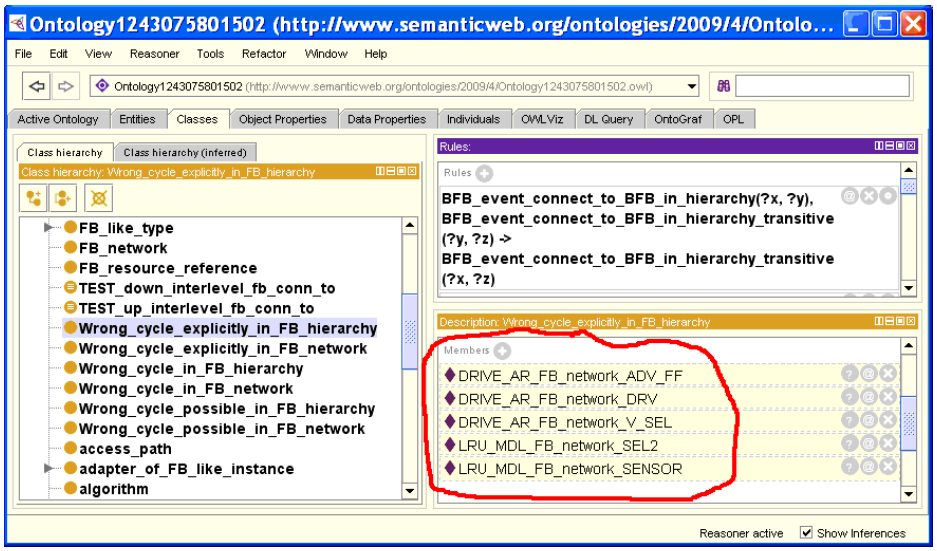


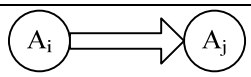
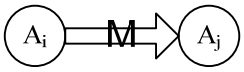
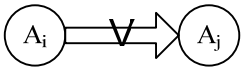
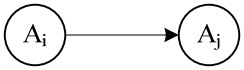
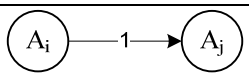
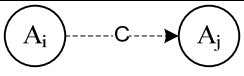
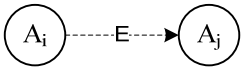
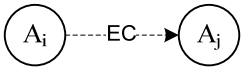
Рис. 7.29. Результаты вычислений в *Protégé* опасных циклов в системе *XBAR_MVC*

7.4. Графы зависимостей для алгоритмов

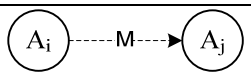
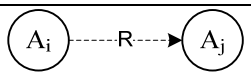
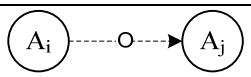
В системах ФБ можно определить граф зависимостей не только на уровне ФБ, но и на уровне алгоритмов (*EC*-акций), входящих в состав базисных ФБ. Алгоритмы можно считать наименьшими базовыми единицами вычислений в системах ФБ. Используя данный граф зависимостей, можно более тонко проанализировать потоки событий и данных, а также их взаимосвязь для определения корректности системы ФБ. Кроме того, данный граф может использоваться для реализации уровней параллельности в системах ФБ на основе концепции опережающих вычислений. Использование концепции опережающих вычислений может использоваться как ключевой момент для параллельной обработки последовательных по своей сути процессов. В табл. 7.7 представлены все виды зависимостей по данным и по управлению, возможные в системах ФБ на уровне алгоритмов. Все приведенные в таблице зависимости могут быть выражены в онтологии ФБ.

Таблица 7.7

Типы связей между алгоритмами и их обозначения

Графическое обозначение	Мнемоническое обозначение	Тип связи
	D	Внутриблочная связь по данным базисного ФБ
	DM	Межблочная связь по данным в пределах одного ресурса
	DV	Межресурсная связь по данным
	CA	Безусловная связь по управлению внутри *EC*-состояния базисного ФБ
	CU	Безусловная связь по управлению между *EC*-состояниями базисного ФБ
	CC	Условная связь по управлению между EC-состояниями базисного ФБ
	CE	Событийная связь по управлению между *EC*-состояниями базисного ФБ
	CEC	Условно-событийная связь по управлению между *EC*-состояниями базисного ФБ

Окончание табл. 7.7

Графическое обозначение	Мнемоническое обозначение	Тип связи
	CM	Межблочная связь по управлению
	CR	Межресурсная связь по управлению
	CO	Механическая (физическая) связь по управлению через объект управления

7.5. Вопросы сложности логического вывода

Сложность логического вывода существенно зависит от типа используемых выражений ДЛ. Если в запросе используются все возможности дескриптивной логики *SROIQ*, которая лежит в основе языка *OWL* 2, то ризонинг будет *2NExpTime*-полным [159]. Как правило, в выражениях ДЛ используются не все возможности, что будет существенно понижать сложность вычислений. Например, если для описания достаточно выразительной силы языка *OWL* 1, основанном на логике *SHOIN*, то сложность ризонинга уменьшится до *NExpTime*. Для точного определения сложности выполнения ДЛ можно воспользоваться интерактивным *Web*-ресурсом «*Complexity of reasoning in Description Logics*» [108].

Все стандартные задачи, решаемые при классификации, для языка *SWRL* неразрешимы. Один из путей достижения разрешимого подъязыка *SWRL* является использование *DL*-безопасных правил [184]. При этом сложность вычислений в комбинации *SROIQ* + «*DL*-безопасные правила» не будет превышать сложность вычислений в логике *SROIQ*.

7.6. Система семантического анализа проектов IEC 61499

На основе разработанного выше подхода предлагается структура программных средств для семантического анализа проектов стандарта IEC 61499. Основными компонентами системы являются: 1) терминологический словарь *Tbox*, относящийся к стандарту IEC 61499; 2) транслятор *XML*-описания ФБ в *OWL/SWRL*-представление, включающее, кроме словаря *Tbox* и правил *SWRL*, также соответствующую базу фактов *Abox*; 3) интерфейсная программа, осуществляющая взаимодействие с системой рассуждений (ризнером)

и пользователем (рис. 7.30). В качестве системы рассуждений можно выбрать свободно распространяемый ризнер *Pellet* [198] или *Hermit* [263]. Он поддерживает ДЛ типа *SROIQ* и язык *SWRL* с «ДЛ-безопасной» семантикой. Интерфейсной программой на первый случай может служить система *Protégé* [201]. Разработка собственной интерфейсной программы с использованием *OWL API* позволит более эффективно организовать вычисления.

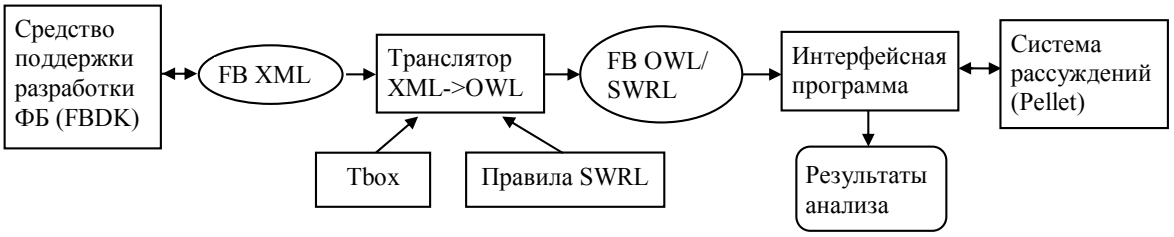


Рис. 7.30. Система семантического анализа проектов IEC 61499 на основе онтологий

По результатам исследований, связанных с семантическим анализом систем управления на основе ФБ IEC 61499, были опубликованы работы [30, 32, 42, 111, 112].

8. Шаблоны модельно-ориентированной реализации систем функциональных блоков стандарта IEC 61499

Международный стандарт IEC 61499 для проектирования распределенных систем управления промышленными процессами определяет абстрактную модель ФБ, допускающую множество различных интерпретаций. Как следствие, в дополнение стандарта были предложены так называемые модели выполнения, определяющие конкретный порядок (дисциплину) выполнения ФБ. Разнообразие моделей выполнения порождает несовместимость различных инструментальных средств, поддерживающих различающиеся модели выполнения, и соответственно проблему портабельности управляющего программного обеспечения (ПО).

Для решения упомянутой выше проблемы портабельности в данном разделе предлагаются *шаблоны реализации* систем ФБ в рамках априори заданных моделей выполнения (или иначе – шаблоны модельно-ориентированной реализации систем функциональных блоков – ШМОРСФБ). Возможно использование другого синонимичного названия – «шаблоны проектирования, устойчивые к семантике выполнения» (*Semantics-Robust Design Patterns*, *SRDP*), введенного в работе [119].

В соответствии со ШМОРСФБ исходная система ФБ корректируется путем включения в ее состав сервисных ФБ, выполняющих ряд функций по интерпретации ФБ, в результате чего поведение системы ФБ в исходной среде выполнения совпадает с поведением исходной системы ФБ в одной, нескольких или всех целевых средах выполнения (рис. 8.1).

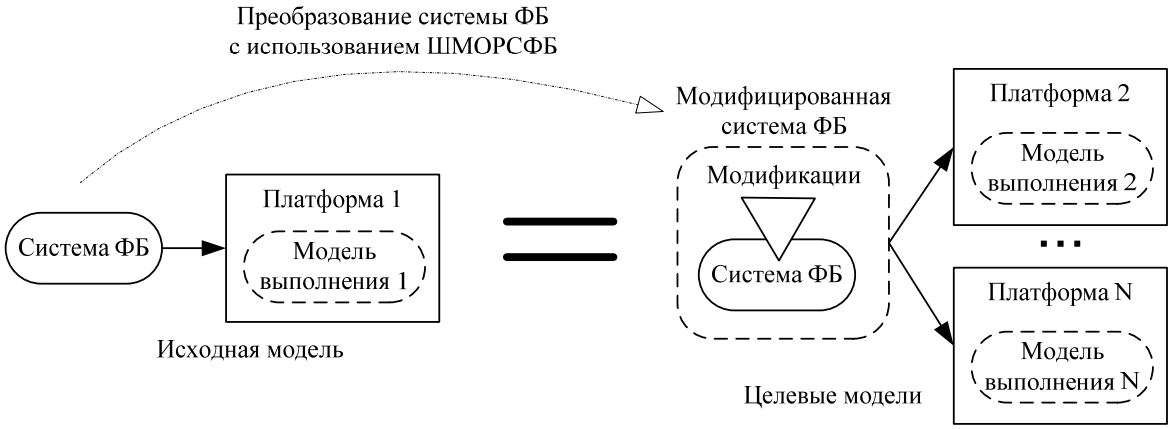


Рис. 8.1. Решение проблемы портабельности с использованием ШМОРСФБ

Концептуально применение ШМОРСФБ состоит из трех шагов: 1) добавление некоторых новых (сервисных) ФБ; 2) изменение некоторых ФБ в исходной системе ФБ; 3) изменение некоторых их взаимосвязей. Функциональные блоки исходной системы и их двойники в результирующей системе ФБ будем в дальнейшем называть *рабочими* ФБ, в то время как ФБ, добавленные в результирующую систему, – *сервисными* ФБ.

ШМОРСФБ можно отнести к специфичным шаблонам проектирования программного обеспечения (ПО) на основе стандарта IEC 61499. В отличие от существующих шаблонов проектирования этого класса (например, *MVC*) ШМОРСФБ не предназначены для использования в ручном проектировании.

ШМОРСФБ могут быть определены как множество принципов, правил, процедур и частичных проектных решений. Предложенные шаблоны реализуются средствами самого аппарата ФБ и поэтому являются достаточно универсальными. Использование аппарата ФБ для определения семантики моделей выполнения ФБ в ряде аспектов сходно с определением семантики UML с помощью некоторого подмножества UML [7]. Здесь также можно провести некоторую аналогию с разработкой онтологии верхнего уровня, относящейся к фундаментальным онтологиям в области концептуального моделирования, например, *Unified Software Modelling Ontology* [88]. В этом случае определяется минимальное множество концептов и отношений, которые необходимы для построения более специфичных доменных онтологий.

Применение ШМОРСФБ может быть значительно упрощено с помощью автоматических трансляторов. Автоматическая трансляция может быть определена и реализована различными способами, в частности, с использованием методов трансформации, основанных на атрибутных графовых грамматиках, предложенных авторами в [246] для рефакторинга приложений IEC 61499. В дальнейшем для определенности и без потери общности в качестве преобразуемых с помощью ШМОРСФБ систем ФБ будем рассматривать приложения (*application*) IEC 61499.

8.1. Общее описание метода

Приложение A_S, спроектированное под известную модель выполнения S (*source*), будет трансформироваться в приложение $A_U = SRDP\ (A_S, S)$, которое будет иметь эквивалентное поведение в любой другой модели выполнения. Здесь индекс U отмечает уни-

версальность. Трансформация состоит из модификации типов ФБ, используемых в A_S, с добавлением в некоторых случаях функционального блока *глобального диспетчера*. Трансформация типов ФБ производится следующим образом.

Для типов базисных ФБ предлагается общий метод трансформации (независимый от исходной модели выполнения *S*), который позволяет определить момент окончания выполнения ФБ при любых входных сигналах, приходящих в любом состоянии, и при любых значениях входных переменных.

Для составного ФБ метод трансформации является специфическим для каждой исходной модели выполнения. В зависимости от этого трансформация может включать добавление следующих ФБ: а) буферов входных сигналов для каждого рабочего ФБ, обеспечивающих доставку всех задействованных входных сигналов, а также желаемый порядок обработки входных сигналов; б) счетчиков квитанций, отмечающих прием входных сигналов буфером. Они нужны для корректного определения завершения работы буфера в системе синхронизации записи сигналов; в) контроллеров передачи сигналов, ответственных за передвижение сигналов между буферами; г) локальных диспетчеров, гарантирующих, что внутри составного ФБ компонентные ФБ будут вызываться в том же порядке, что и в исходной модели.

В целом эти трансформации обеспечивают эквивалентность результирующего поведения системы ФБ и ее оригинального поведения в исходной модели выполнения.

Следует отметить, что если целевая модель выполнения *T* известна, то возможно разработать частный шаблон *SRDP'*(A_S,*S*,*T*), который создает более эффективное приложение A_T, эквивалентное A_S, но только по отношению к целевой платформе *T*. Можно заметить, что аргументы у шаблонов *SRDP* и *SRDP'* различны.

8.2. Трансформации базисных функциональных блоков

8.2.1. Трансформация интерфейсов

Целью трансформации базисных ФБ является добавление в них функциональности, позволяющей сигнализировать об окончании выполнения *ECC* путем выдачи через дополнительный событийный выход *ее* некоторого служебного сигнала, сопровождаемого значе-

нием дополнительной выходной переменной *nOut*, определяющей число сигналов, выданных ФБ за время его прогона. Все это необходимо для корректного определения окончания передачи сигналов из рабочего ФБ в соответствующий буфер. Если ни один выходной сигнал не выдавался, то значение этой переменной равно нулю (*nOut* = 0). Переменная *nOut* служит для исключения «гонок» в системе ФБ, реализуемой с использованием ШМОРСФБ, и ее подробное описание использования будет приведено ниже. Следует отметить, что если целевая модель выполнения использует одноэтапную передачу сигналов между ФБ (например, синхронная или циклическая модели выполнения), то тогда нет необходимости в переменной *nOut*, и завершение записи сигналов в буфер должно определяться самим буфером (без использования переменной *nOut*). Модификация интерфейса базисного ФБ проиллюстрирована на рис. 8.2.

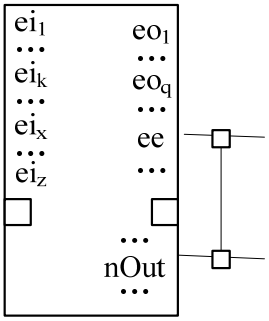


Рис. 8.2. Изменения в интерфейсе
рабочего базисного ФБ
для использования в ШМОРСФБ

8.2.2. Трансформация диаграммы *ECC*

Для того, чтобы достичь «расширенного» поведения базисного ФБ, определенного представленным выше интерфейсом, диаграмма *ECC* должна быть соответствующим образом преобразована.

В рамках предлагаемых ШМОРСФБ сигнал окончания выполнения ФБ (сигнал *ee*) должен выдаваться даже в том случае, когда приходящий входной сигнал не вызывает срабатывания переходов в диаграмме *ECC* (так называемый *незначащий* входной сигнал). По сути дела, сигнал *ee* должен отмечать переход *OSM* в состояние *s*0 (переход *t*2) [163].

Правила трансформации *ECC* будут представлены с использованием нотации атрибутных графовых грамматик и нотации раздела 6 для представления дуг диаграммы *ECC*.

Трансформация *ECC* будет выполнена за четыре шага. Во-первых, будет проведен рефакторинг *ECC* для избавления от *EC*-состояний, из которых выходят *C*- и *E*-дуги. В результате множество *EC*-состояний разбивается на два множества – множество терминальных *EC*-состояний, из которых выходят только *E*-дуги, и промежуточных (транзитных) состояний, из которых выходят только *C*-дуги. Как показано в работе [246], это возможно в случае, когда сторожевые условия зависят от предшествующих *EC*-акций. Во-вторых, во все терминальные *EC*-состояния добавляется выдача сигнала *ee*. Третья трансформация применяется к терминальным *EC*-состояниям, чтобы гарантировать, что сигнал *ee* будет выдаваться для всех входных сигналов и при любых значениях переменных. Цель четвертой трансформации – подсчет числа выходных сигналов, выдаваемых ФБ. Для более глубокого понимания семантики правил можно представить их левые и правые части в форме сети *XNet* [148]. Однако в данном случае считается, что модель выполнения *ECC* является неизменной, она хорошо известна и находится в практическом использовании, следовательно, правила, приведенные ниже, вполне очевидны с логической точки зрения и не требуют дополнительных семантических объяснений.

Шаг 1. *Разделение терминальных и транзитных EC-состояний*

EC-состояние называется *терминальным*, если из него выходят только *E*-дуги. Когда происходит переход в терминальное *EC*-состояние, ФБ завершает свое выполнение и ожидает приема новых сигналов. Таким образом, сигнал *ee* должен выводиться только в терминальных *EC*-состояниях.

Правило трансформации на рис. 8.3, дополняющее набор правил для рефакторинга из раздела 6, устраняет те *EC*-состояния, из которых одновременно выходят как *E*-дуги, так и *C*-дуги. Для этого вводится дополнительное состояние s_a и дуга (s_i, s_a), имеющая сторожевое условие $\sim c_k \& \ldots \& \sim c_n$, определяющее явный переход из *EC*-состояния s_i, где все сторожевые условия на исходящих *C*-дугах оценены как ложные. Генерируемое *EC*-состояние s_a является терминальным.

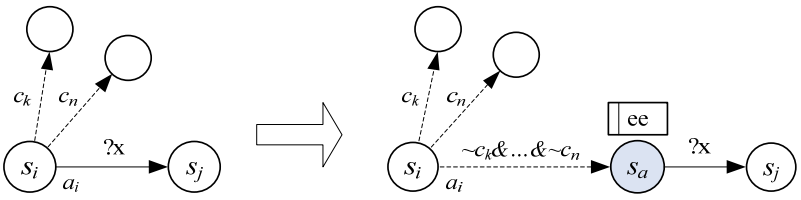


Рис. 8.3. Правило для определения терминального *EC*-состояния в случае выходящих *E*- и *C*-дуг

Шаг 2. *Добавление выдачи сигнала ее в терминальных состояниях*

Выдача выходного сигнала *ее* добавляется ко всем терминальным *EC*-состояниям. В этом случае сигнал *ее* будет выдаваться в конце каждого прогона ФБ. На рис. 8.3 показано совместное применение трансформаций 1 и 2.

Шаг 3. *Выдача сигнала ее для всех входных событий и всех входных данных*

Трансформированная на шагах 1–2 диаграмма *ECC* не может выдать сигнал *ее*, если ФБ принимает *незначащий* входной сигнал, который не порождает изменение состояния. Это может произойти вследствие того, что сигнал не входит ни в одно из условий EC-переходов, исходящих из текущего *EC*-состояния, или если нет разрешенных сторожевых условий исходящих *EC*-переходов. Предлагаемая трансформация решает эту проблему путем модификации дуг, исходящих из терминального *EC*-состояния так, что *EC*-переход срабатывает даже при незначащем входном сигнале. Обобщенное правило трансформации приведено на рис. 8.4. На данном рисунке символами *ei*, *eo*, *a* с индексами обозначены соответственно событийные входы, событийные выходы и алгоритмы. Сторожевые условия, связанные с событийным входом ei_k, обозначены как c_{k1}, c_{k2}, …, c_{kn}.

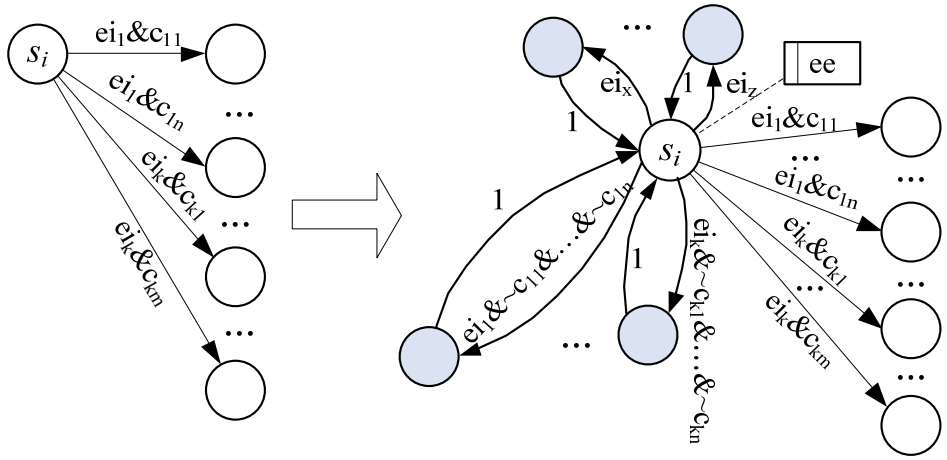


Рис. 8.4. Правило, обязывающее сработать какой-либо *EC*-переход в терминальном *EC*-состоянии

EC-состояние s_i дополняется двухреберными циклами (для краткости назовем их петлями), реализующими сохранение состояния s_i. Число таких петель равно числу событийных входов ФБ. Каждая петля состоит из *EC*-перехода в некоторое вспомогательное *EC*-состояние (обозначенное залитым кружком) и обратного (всегда

истинного) *EC*-перехода в s_i. Существует два вида условий на дугах этих двухреберных циклов:

1) условие первого вида образуется из одного из входных сигналов, помечающих исходящую дугу, в совокупности с конъюнкцией, объединяющей отрицания сторожевых условий, сопровождающих данный входной сигнал на исходящих дугах. Число циклов такого рода равно числу отличающихся входных сигналов, отмечающих исходящие дуги. На рис. 8.4 это входные сигналы (ei_1, …, ei_k);

2) условие второго вида образуется из входных сигналов, обработка которых не предусмотрена в рассматриваемом состоянии. Иными словами, сигналы из этого множества не входят в условия исходящих из состояния *EC*-переходов. На рис. 8.4 справа это входные сигналы (ei_x, …, ei_z).

Шаг 4. *Подсчет числа выходных сигналов, выданных ФБ*

На первый взгляд эта задача тривиально решается дополнением каждого алгоритма *EC*-акции, в которой выдается выходной сигнал, оператором инкремента счетчика *nOut*. Этого можно достичь введением новой специальной *EC*-акции A_I, выполняющей эту функцию. Однако ситуация осложняется возможным появлением петель в терминальных *EC*-состояниях, которые возникают в результате применения правила шага 3. Правило на рис. 8.5 решает эту проблему путем передвижения *EC*-акций из «неподходящего» терминального *EC*-состояния s_i в промежуточное *EC*-состояние s_a и расширения *EC*-акций инкрементированием *nOut*. Это необходимо, поскольку сигналы на петлях в *EC*-состоянии s_i могли бы вызвать те же самые действия, что привело бы к некорректному поведению.

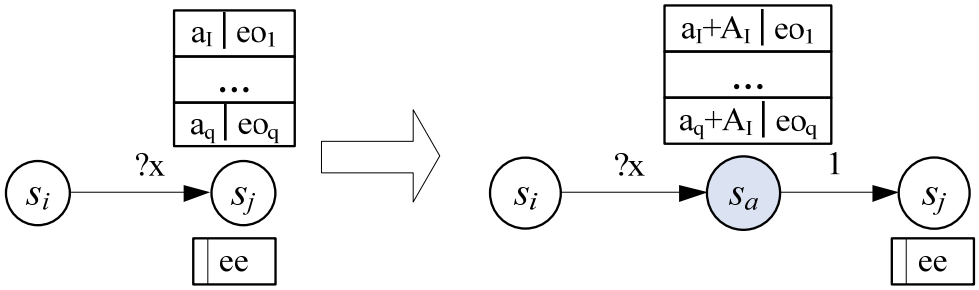


Рис. 8.5. Правило для передвижения *EC*-акций
из неподходящего *EC*-состояния и добавления инкремента *nOut*

Сброс счетчика *nOut* можно было сделать в первом операторе алгоритма первой *EC*-акции, присоединенной к целевому *EC*-состоянию *E*-дуги, исходящей из терминального *EC*-состояния. Этого можно также добиться введением новой специальной *EC*-акции A_0,

выполняющей эту функцию. Правило, добавляющее A_0 к *EC*-акции для сброса счетчика *nOut*, показано на рис. 8.6.

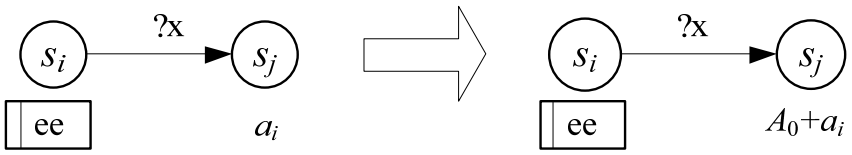


Рис. 8.6. Правило для введения *EC*-акции A_0 для сброса счетчика *nOut*

Представленные трансформации *ECC* сохраняют исходное поведение, но добавляют выдачу сигнала *ee*, сопровождаемого числом выданных событий *nOut*. Эквивалентность может быть строго доказана, но доказательства опущены вследствие ограниченности печатного места и интуитивной ясности трансформационных свойств.

Предложенные трансформации являются развитием метода рефакторинга *ECC*, рассмотренного в разделе 6. Для каждого из правил на рис. 8.3–8.6 в общем случае было разработано несколько правил системы трансформации графов *AGG* [75], которые были добавлены в множество правил существующей системы рефакторинга *ECC*. Результирующая база правил была протестирована в *AGG* на нескольких примерах.

8.2.3. Пример. Трансформация блока *CruiseController*

Рассмотрим пример трансформации базисного ФБ *CruiseController*. Этот ФБ является основной частью системы автоматического поддержания скорости движения (системы круиз-контроля), представленной в [257]. Для повышения читабельности рисунков введем краткое обозначение элементов *ECC* из [257]. Сигналы e_0, e_1 и e_2 будут представлять сигналы *INIT* (инициализация), *SpeedChange* (изменение скорости) и *DesiredSpeedChange* (изменение требуемой скорости) соответственно. Имена событийных выходов (одноименных с соответствующими *EC*-акциями) *INITO* (подтверждение инициализации), *ThrottleOff* (отключить дроссельную заслонку), *ThrottleUp* (открыть дроссельную заслонку) и *ThrottleDown* (закрыть дроссельную заслонку) сокращаются до a_0, a_1, a_2, и a_3 соответственно.

Для сторожевых условий используются следующие сокращения:

$c_1 \equiv CurrentSpeed=DesiredSpeed$;

$c_2 \equiv CurrentSpeed<DesiredSpeed$;

$c_3 \equiv CurrentSpeed>DesiredSpeed$;

$c_4 \equiv DesiredSpeed=-1$; $c_5 \equiv DesiredSpeed>-1$,

где *CurrentSpeed* и *DesiredSpeed* – текущая и требуемая скорости автомобиля соответственно.

Трансформация интерфейса представлена на рис. 8.7, исходная *ECC* – на рис. 8.7,*б*, диаграмма *ECC* после рефакторинга – на рис. 8.7,*в* и результирующая *ECC* – на рис. 8.7,*г*.

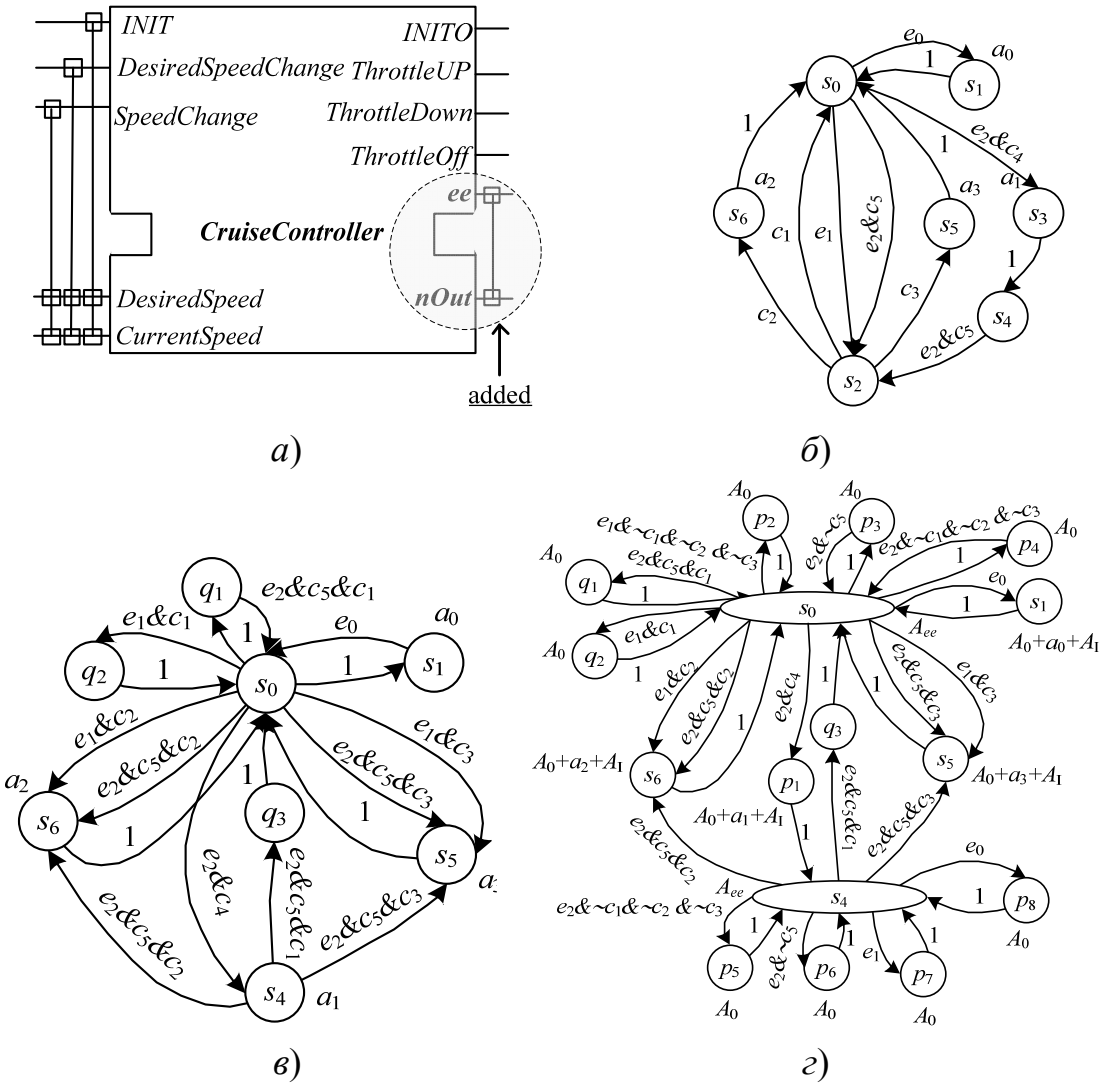


Рис. 8.7. Трансформации ФБ *CruiseController*:
а – трансформированный интерфейс; *б* – исходная *ECC*;
в – *ECC* после рефакторинга; *г* – трансформированная *ECC*

В процессе трансформации некоторые *EC*-состояния могут быть уничтожены (например, *EC*-состояние s_2 было удалено во время рефакторинга), в то же время могут быть созданы некоторые новые *EC*-состояния (эти состояния имеют идентификаторы *p* и *q* с индексами). На рис. 8.7,*г*, A_{ee} означает *EC*-акцию, выводящую сигнал *ee*.

8.3. Буферирование сигналов

Буферирование сигналов требуется в реализации нескольких моделей выполнения, например, в последовательной модели, реализованной в *FORTE* [264]. Поэтому наряду с введенной модификацией базисного ФБ другим важным реализационным механизмом ШМОРСФБ является буферирование сигналов. Управляемый буфер определяет правила передачи, буферирования и обработки входных сигналов. Выбор и реализация такого буфера зависят от исходной модели выполнения, а именно, от свойств буферизации, которые необходимо сохранить при переносе приложения в целевую модель выполнения. Следует отметить, что некоторые семантики выполнения могут не требовать буферов изначально, но поддержка одного и того же механизма планирования событий на целевой платформе будет определять эту необходимость. В качестве примера можно привести воспроизведение правил синхронной модели в последовательной семантике.

Можно реализовать буфера, поддерживающие различные дисциплины буферирования (например, *FIFO*, *LIFO* на основе приоритетов), а также сохранение множественных сигналов для организации очередей сигналов на событийных входах ФБ. Однако в дальнейшем будет использоваться *FIFO*-буфер с сохранением только одного сигнала на одном событийном входе, поскольку это наиболее согласованный со стандартом IEC 61499 и большинством моделей выполнения вариант.

Интерфейс ФБ *Buffer* изображен в левой части рис. 8.8. Входы ei_1, ..., ei_n соответствуют входным событийным линиям, через которые сигналы приходят и помещаются в буфер. Выходные событийные линии eo_1, ..., eo_n используются для вывода соответствующих сигналов из буфера. Запись входного сигнала ei_k в буфер подтверждается соответствующим выходным сигналом ack_k. При приеме входного сигнала-запроса *putOut* из буфера выводится ровно один выходной сигнал. Если при этом буфер пуст, то выводится выходной сигнал *empty*.

Буфер сигналов может быть выполнен в виде кольцевого буфера. Он может быть эффективно реализован (например, как СИФБ) с использованием трех простых и быстрых операций: инкремента индекса, прямой записи (чтения) идентификатора событийного входа в (из) буфер и сравнения с верхней границей базового массива. Диаграммы временных последовательностей, описывающие работу буфера, показаны в правой части рис. 8.8.

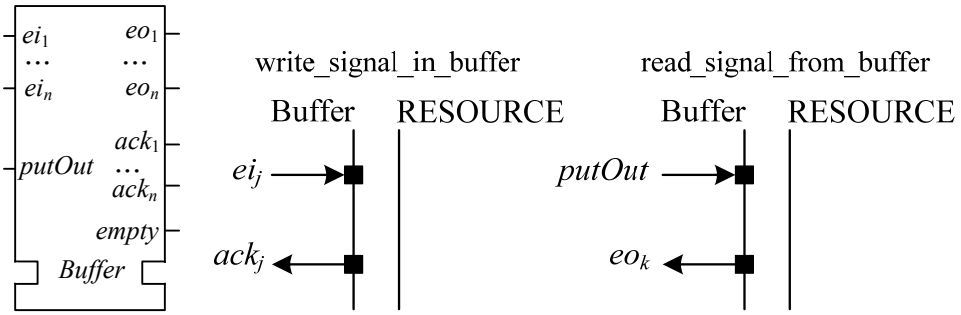


Рис. 8.8. ФБ для буфера:
слева – интерфейс, справа – диаграммы временных последовательностей

Для оценки сложности, вносимой использованием буферов, предположим, что время выполнения этих операций одинаково и равно t_1. Поскольку каждый сигнал сначала пишется, а затем читается из буфера, задержка в передаче каждого сигнала между парами ФБ (с учетом того, что только один буфер между ними) равна $6 \times t_1$, в то время как передача n сигналов требует $6 \times n \times t_1$ единиц времени.

Как будет видно в последующих разделах, существуют две типичные структуры взаимосвязи буферов, возникающие при использовании ШМОРСФБ. Назовем их *inb* и *outb* соответственно (рис. 8.9). Например, в структуре *inb* все сигналы из входного буфера buf_i последовательно обрабатываются блоком fb_i и записываются в выходные буфера buf_j, ..., buf_k.

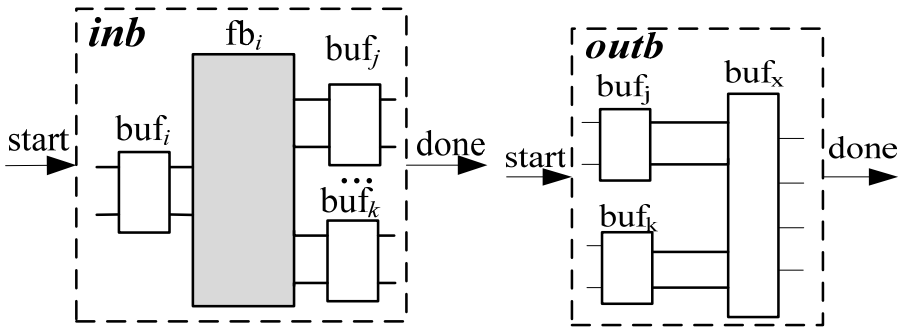


Рис. 8.9. Структуры *inb* и *outb* потоков событий

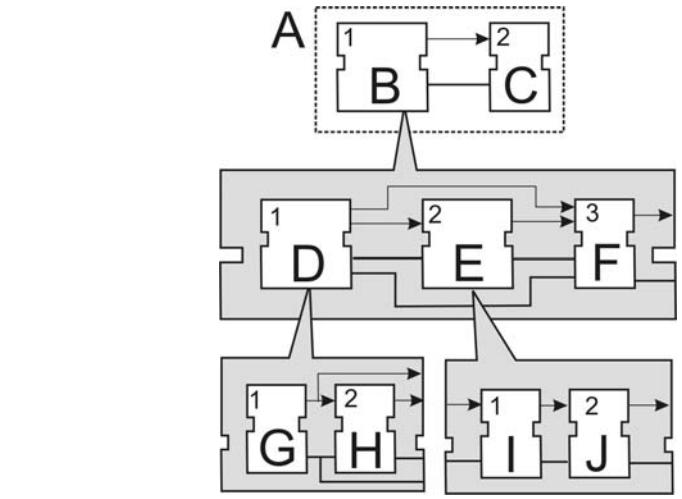
8.4. Трансформация составных функциональных блоков

В данном разделе рассматривается концепция применения ШМОРСФБ в составных ФБ на примере циклической и синхронной моделей выполнения.

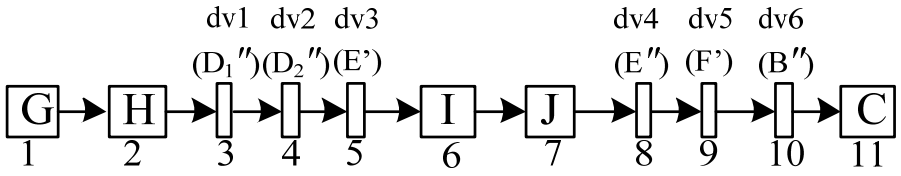
8.4.1. Циклическая модель выполнения

Существует несколько моделей выполнения составных ФБ, а именно: как единой сущности и как контейнера [218]. В данной работе будем придерживаться второй модели. Как было показано в разделе 3, многоуровневую систему ФБ, содержащую базисные, составные и сервисные интерфейсные ФБ, можно свести к одноуровневой системе, содержащей только базисные и СИФБ, а также так называемые клапаны данных (КД), представляющие интерфейсную логику. Клапаны данных могут быть эффективно реализованы в виде СИФБ.

Циклическая модель выполнения предполагает, что порядок выполнения ФБ в системе явно задан. *Глобальный* порядок выполнения может быть однозначно определен на основе *локальных* порядков в составных ФБ. Это проиллюстрировано на рис. 8.10 на примере абстрактной иерархической системы ФБ, где *A*, *B*, *D* и *E* – составные ФБ, а *C*, *F*, *G*, *H*, *I* и *J* – базисные ФБ.



а)



б)

Рис. 8.10. Иерархическая система ФБ
с локальными порядками выполнения (*а*) и ее «плоское» представление
с глобальным порядком выполнения, назначенным ФБ и КД (*б*)

Цифрами обозначен локальный порядок выполнения блоков внутри составного ФБ. Глобальный порядок выполнения может быть определен в виде следующей последовательности: *G*, *H*, D_1",

D_2", E', I, J, E'', F, B'', C. Здесь символом со штрихом обозначена активность входной интерфейсной логики соответствующего ФБ, а символом с двумя штрихами – активность выходной интерфейсной логики. Функциональный блок D имеет два событийных выхода, поэтому используются два отдельных клапана данных D_1" и D_2".

Поскольку активность по выполнению составного ФБ состоит в основном из активностей выполнения входящих в его состав базисных ФБ, то сосредоточимся в основном на базисных ФБ. В соответствии с заданными локальными порядками выполнения глобальный порядок выполнения базисных ФБ будет следующим: G, H, I, J, C.

На рис. 8.10,*б* показано одноуровневое («плоское») представление той же самой системы. Возникшие КД обозначены как $dv1$–$dv6$. Числа под ФБ и КД обозначают глобальный порядок выполнения. Как видно из рисунка, клапаны данных диспетчируются единообразно с остальными ФБ системы. Плоская структура может быть использована для достижения поведения, эквивалентного циклическому, в любой другой модели поведения, так как она сохраняет порядок выполнения исходной системы.

Другой подход к применению ШМОРСФБ к составным ФБ основан на использовании «родной» интерфейсной логики составного ФБ без приведения системы к одноуровневому представлению. Способ трансформации содержимого составного ФБ зависит от того, является ли этот блок базисным или составным. Правила, которыми надо руководствоваться при реализации составных ФБ в циклической модели поведения, следующие:

1) для базисного компонентного ФБ создаются по одному входному буферу сигналов, а также счетчик квитанций записи в выходные буфера;

2) для составного компонентного ФБ дополнительные (инфраструктурные) блоки не создаются;

3) в случае, когда ФБ передает сигналы в буфер, расположенный в другом ФБ, счетчик квитанций рекомендуется располагать в том ФБ, где находится ФБ-источник сигналов;

4) если не предполагается перенастройка порядка выполнения компонентных ФБ в рамках объемлющего ФБ, то заданный порядок выполнения можно обеспечить последовательным соединением этих ФБ;

5) главный диспетчер включается только на уровне ресурса (т.е. в систему ФБ верхнего уровня иерархии). Например, относительно рис. 8.9 можно сказать, что диспетчер должен располагаться только в системе A.

Концептуальная трансформация составного ФБ типа *B* из рис. 8.10 показана на рис. 8.11. Два из компонентных ФБ в блоке *B* являются составными (*D* и *E*) и один – базисным (*F*). Штриховым контуром обведена группа базисного ФБ. Блок *buf* представляет буфер сигналов, а блок *cOut* – счетчик квитанций. Цифрами в левой части рисунка определен локальный порядок выполнения компонентных ФБ в рамках объемлющего ФБ. Для последовательного выполнения ФБ в данном случае используется соединение ФБ в цепочку, а не диспетчер. Данный способ использовался, например, в работе [265]. Преимуществом такого способа является простота, а недостатком – «жесткость» структуры и трудоемкость ее перенастройки.

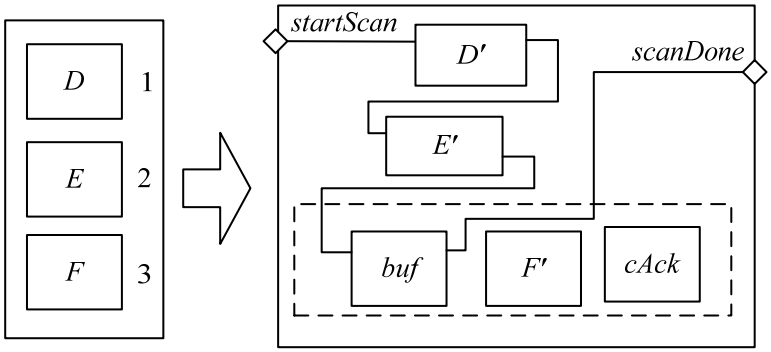


Рис. 8.11. Пример преобразования составного ФБ промежуточного уровня, содержащего составные и базисные компонентные ФБ, в рамках циклической модели выполнения

На рис. 8.12 показаны изменения, вносимые в интерфейс рабочего составного ФБ. Вновь появившиеся входы/выходы отмечены подведенными к ним отрезами прямых. Как видно из рис. 8.12, появился событийный вход *startScan*, запускающий однократный прогон ФБ, входящих в состав данного ФБ, а также событийный выход *scanDone* для определения окончания данного однократного прогона. Локальный порядок выполнения ФБ в составе составного ФБ определяется с помощью входной переменной – списка опроса *exList*. Ее инициализация осуществляется сигналом на событийном входе *INIT*. Следует, однако, заметить, что при использовании диспетчера с «жесткой» логикой (как, например, на рис. 8.11) необходимость во входах *INIT* и *exList* отпадает. Событийные входы $acki_1$, ..., $acki_n$ используются для приема информации о записи сигналов в буфера, лежащие за границами данного ФБ. В то же время через событийные выходы $acko_1$, ..., $acko_n$ внешним счетчикам квитанций выдаются сигналы квитирования записи в локальные буфера.

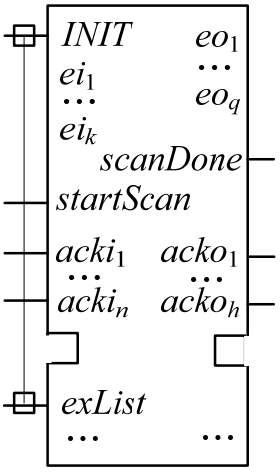


Рис. 8.12. Изменения в интерфейсе рабочего составного ФБ для использования его в рамках шаблона реализации «Циклическая модель выполнения»

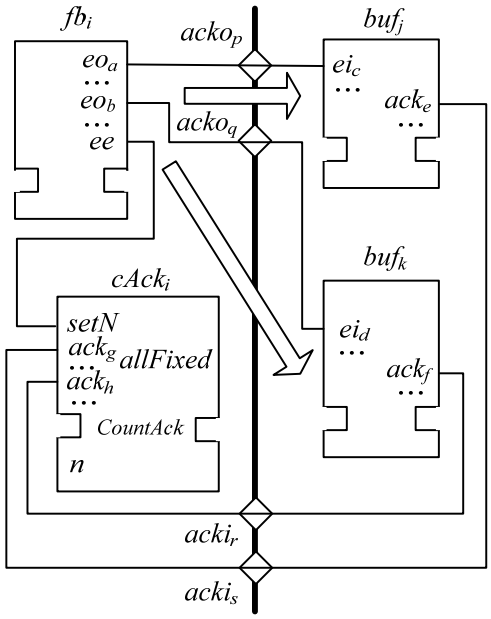


Рис. 8.13. Схема квитирования записи сигналов в буфер, лежащий вне составного блока

Более подробно схема подобного квитирования через интерфейс представлена на рис. 8.13. На данном рисунке толстой вертикальной линией обозначен межуровневый интерфейс, т.е. интерфейс между объемлющим и раскрытым компонентным ФБ. Функциональный блок $cAck_i$ в данном случае является счетчиком квитанций. Направления передач сигналов между ФБ показаны широкими стрелками.

8.4.2. Синхронная модель выполнения

Как было отмечено в разделе 1, существует несколько разновидностей синхронной модели выполнения. Далее будет рассматриваться двухфазная синхронная модель с грануляцией выполнения на уровне ФБ. Для определенности будем считать, что целевое приложение будет выполняться последовательно. Таким образом, рассматриваемый шаблон будет эмулировать синхронность и параллелизм исходного приложения.

Основная проблема в реализации составных ФБ в рамках синхронной модели выполнения заключается в реализации второй фазы, состоящей из передачи сигналов из выходных буферов ФБ во входные буфера в случае, когда буфера-источники и буфера-приемники находятся в разных объемлющих ФБ, принадлежащих

разным уровням или разным веткам системной иерархии. На первой фазе подобных проблем нет, поскольку все ее участники (входной буфер, сам ФБ и выходной буфер) локализованы в рамках одного объемлющего блока.

Упомянутая выше проблема показана на рис. 8.14, построенным на основе рис. 8.9, для случая, когда все буфера лежат в разных объемлющих ФБ *fb*1, *fb*2 и *fb*3 соответственно и, кроме того, в разных ветках системной иерархии. На рис. 8.14 с целью упрощения явно не показаны событийно-потоковые связи между буферами, вместо этого широкими стрелками обозначены направления передач сигналов. В дополнение к буферам сигналов в структуру был добавлен контроллер перемещения сигналов d_x типа *dispOutMoving*, который реализует перемещение сигналов между выходными буферами buf_j и buf_k блоков fb_a и fb_c соответственно и входным буфером buf_x блока fb_x. Как можно увидеть из рис. 8.14, в результате «децентрализации» структуры *outb* появляется несколько разрывов событийных связей, каждый из которых оформляется в виде событийных входов/выходов интерфейсов соответствующих ФБ.

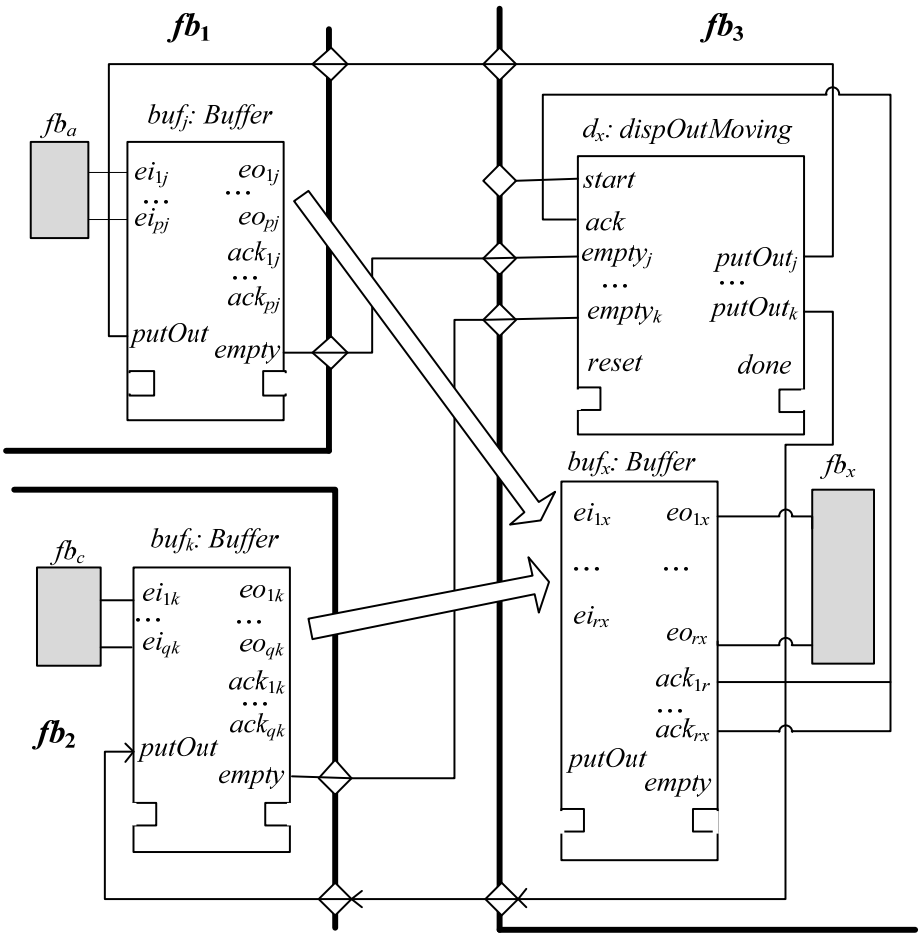


Рис. 8.14. Детализированная децентрализованная структура *outb*, когда все буфера находятся в разных объемлющих ФБ

Трансформация содержимого составного ФБ будет зависеть от того, является ли этот блок базисным или составным. Правила, которыми надо руководствоваться при реализации составных ФБ в синхронной модели поведения, следующие:

1) для базисного компонентного ФБ создаются по одному входному и выходному буферу сигналов, а также счетчик квитанций записи в выходные буфера;

2) для составного компонентного ФБ, входящего в состав объемлющего ФБ, дополнительные (инфраструктурные) блоки не создаются;

3) контроллер перемещения сигналов типа *dispOutMoving* (если он необходим) располагается в том ФБ, где находится принимающий входной буфер;

4) в содержимое составного ФБ промежуточного уровня включается пара блоков типов *gather*1 и *gather*2, функцией которых является сбор подтверждений от структур типов *inb* и *outb* об окончании их выполнения и формировании выходных сигналов об окончании выполнения первой и второй фаз в рамках данного составного ФБ. Блоки типов *gather*1 и *gather*2 можно считать усеченными диспетчерами, функция которых сведена к управлению однократным выполнением одной фазы в локальной области;

5) главный диспетчер включается только в систему ФБ верхнего уровня, располагаемую на ресурсе.

Работа каждой из фаз (первой и второй) в рамках целой системы складывается из работ по выполнению ФБ, проводимых в локальных областях (в составных ФБ). Иерархически взаимосвязанная система из блоков *gather*1 (*gather*2) образует единую систему определения окончания работы подсистем типа *inb* (*outb*) в глобальном масштабе.

На рис. 8.15 схематично приведен пример преобразования составного ФБ промежуточного уровня, содержащего составные (*D* и *F*) и базисный (*F*) компонентные ФБ. Штриховым контуром обведена группа базисного ФБ. Для простоты считается, что базисный ФБ *F* имеет в качестве предшественника только один базисный ФБ, поэтому блок управления перемещением сигналов не требуется. Хотя очевидно, что результирующий ФБ сложнее оригинального ФБ, однако сложность, вносимая сервисными ФБ, интуитивно такого же порядка, что и у исходного приложения.

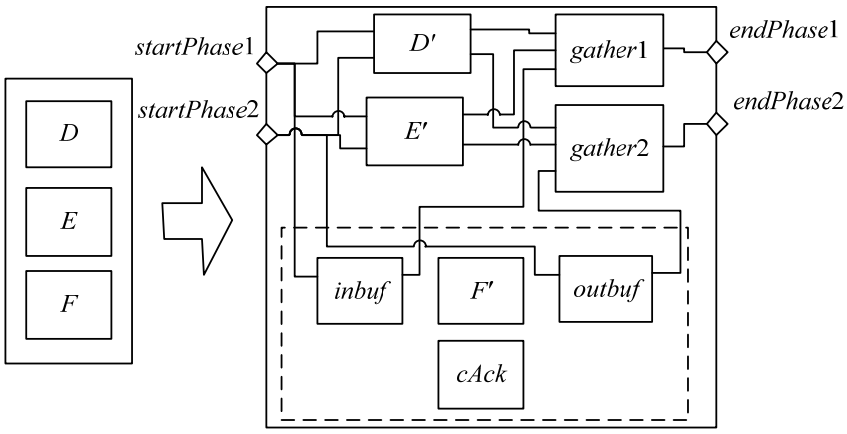


Рис. 8.15. Пример преобразования составного ФБ промежуточного уровня, содержащего составные и базисные компонентные ФБ, в рамках синхронной модели выполнения

8.5. Трансформация СИФБ

В данном разделе трансформация СИФБ в соответствии с шаблонами реализации детально не рассматривается. Тем не менее можно предположить, что СИФБ – источник сигналов, который должен быть связан с выходным буфером, а СИФБ-приемник – с входным буфером. На рис. 8.16 представлен вариант использования СИФБ первого типа. При этом СИФБ соединяется с буфером сигналов типа *BufferAll*. Этот буфер является своего рода представителем (*proxy*) СИФБ. Он накапливает сигналы, приходящие с выходов СИФБ, а по сигналу *start*, исходящему из диспетчера, выдает все накопленные сигналы. При этом (так же, как и в базисном ФБ) по завершению выдачи всех сигналов генерируется сигнал *ee* об окончании работы буфера-посредника, а также число выданных сигналов *nOut*.

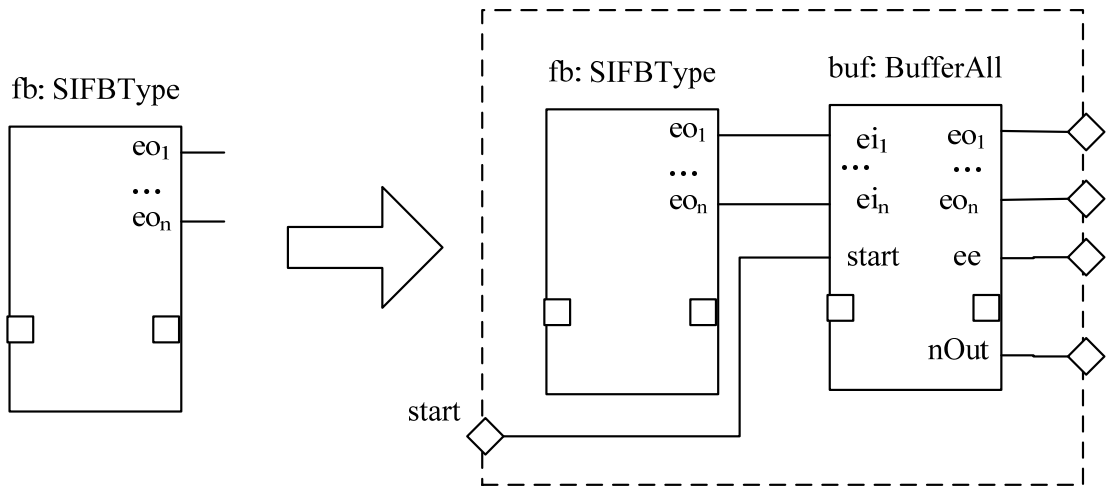


Рис. 8.16. СИФБ с буфером-посредником для использования в ШМОРСФБ

8.6. Шаблон «Циклическая модель выполнения»

В качестве иллюстративного «ссылочного» примера системы ФБ будем рассматривать сеть ФБ, приведенную на рис. 8.17. Все связи между ФБ являются событийными. В приведенной структуре встречаются как развилки потоков событий, так и их слияния.

Схема буферирования шаблона «Циклическая модель выполнения» проиллюстрирована (на основе системы ФБ из рис. 8.17) на рис. 8.18. В дальнейшем будем четко выделять входные и выходные буфера ФБ. Как видно из рис. 8.18, каждому ФБ ставится в соответствие (входной) буфер сигналов. Число событийных входов и выходов этого буфера равно числу событийных входов ФБ-владельца. Событийные выходы буфера связываются с событийными входами ФБ соответственно. Следует отметить, что на рис. 8.18 показаны не все аспекты трансформации одной сети в другую, указываются только сгенерированные буфера данных и потоки событий между элементами структуры. Число структур типа *inb* в результирующей системе ФБ равно числу ФБ в «ссылочной» системе ФБ. В структуре *inb* все события из входного буфера buf_i последовательно обрабатываются в блоке fb_i, выходные сигналы записываются в выходные буфера buf_j, ..., buf_k.

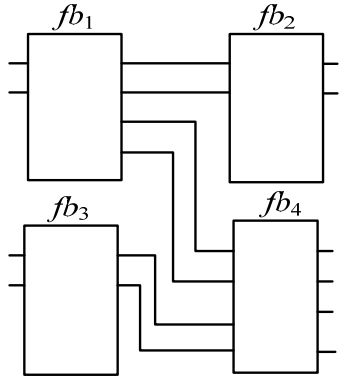


Рис. 8.17. Система ФБ, используемая в качестве примера для реализаций с использованием различных шаблонов

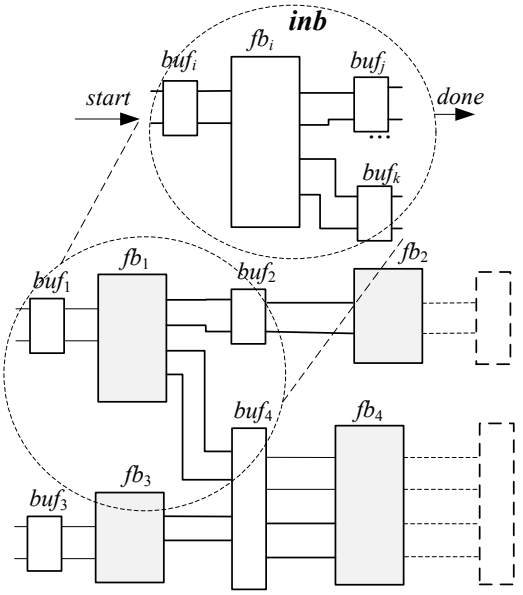


Рис. 8.18. Буферирование в системе ФБ из рис. 8.17, реализованной в рамках циклической модели выполнения

На рис. 8.19 приведена детализированная структура *inb*.

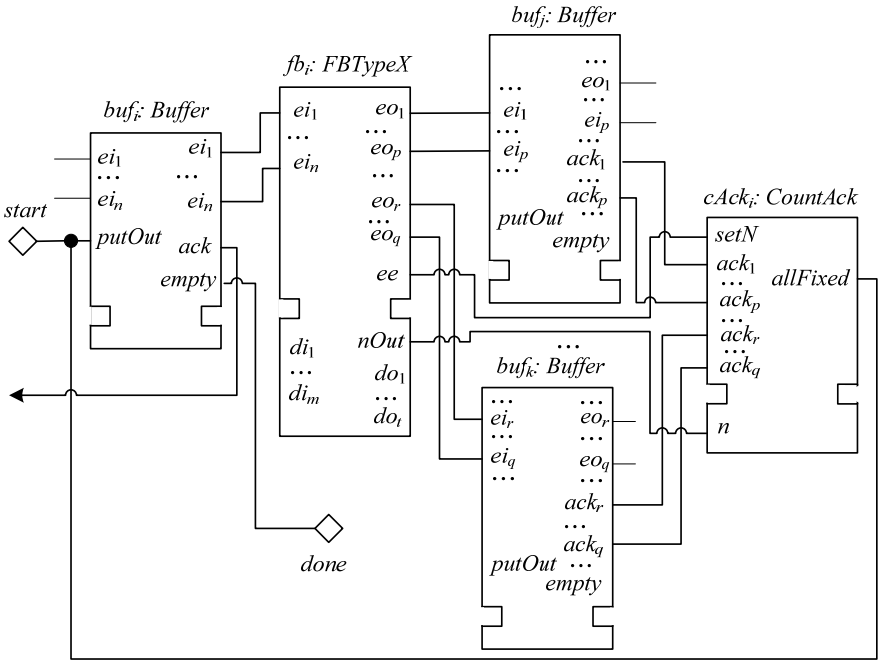


Рис. 8.19. Детализированная структура *inb*

Как видно из данного рисунка, каждому ФБ (помимо входного буфера) ставится в соответствие счетчик числа подтверждений фиксаций сигналов в выходных буферах (ФБ типа *CountAck*). Механизм подтверждений необходим в данном случае для того, чтобы четко обозначить окончание фазы обработки сигнала из входного буфера, включая запись всех выходных сигналов данного прогона ФБ. Отсутствие такого механизма синхронизации могло бы привести в некоторых целевых моделях выполнения к некорректному функционированию системы.

Следует отметить, что не все событийные выходы типа *ack* выходных буферов для *i*-го блока связаны со счетчиком квитанций, ассоциированным с этим ФБ, а только те, которые отражают работу данного блока, т.е. являются связанными с выходными сигналами, исходящими из блока. Данное положение схематично проиллюстрировано на рис. 8.20. На данном рисунке представлена ситуация, когда несколько блоков (в данном случае два – блок fb_1 и fb_2) связаны с одним (выходным) буфером buf_3. Сплошными линиями представлены потоки событий между ФБ, пунктирными – линии подтверждений записи в буфер (с выходов *ack* буферов), а штрихпунктирной – линии окончания работы ФБ. Как можно заметить из рисунка, линии подтверждений делятся на группы согласно принадлежности событийных линий тому или иному ФБ.

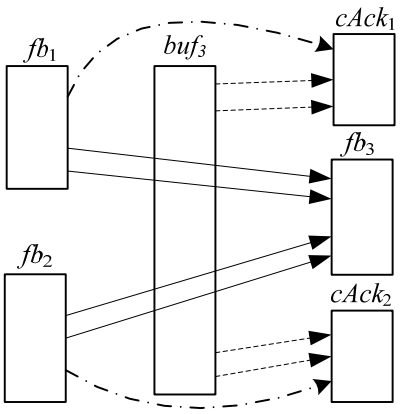


Рис. 8.20. Схема подсчета квитанций,
когда несколько ФБ связаны с одним буфером

Следует также заметить, что приведенная на рис. 8.19 схема может применяться только в системах ФБ, где нет неявного расщепления сигналов. При наличии расщепления счетчик подтверждений (ФБ типа *CountAck*) будет работать неправильно, поскольку он считает число реальных записей в буфера, в то время как переменная *nOut* при завершении работы ФБ представляет число выданных сигналов *без учета расщепления*.

Можно предложить два пути решения «проблемы расщепления сигналов». Первый способ состоит в замене в исходной системе ФБ неявного расщепления явным с использованием блока *E_SPLIT*. Второй способ предполагает выдачу скорректированного значения *nOut* с учетом размножения сигналов. Недостаток первого способа заключается в увеличении структурной сложности схемы (а значит, и быстродействия). У второго способа этот недостаток отсутствует. Однако в этом случае при «рефакторинге» базисных ФБ следует учитывать контекст их применения, а именно: число выходящих линий из каждого событийного выхода для того, чтобы значение переменной *nOut* представляло реальное число выдаваемых блоком сигналов с учетом их (конкретного) расщепления, что является недопустимым.

Следует заметить, что в представленной на рис. 8.19 схеме возможны «гонки» сигналов. Это ситуация, когда из некоторого ФБ-источника имеется несколько путей передачи сигналов (через разные ФБ) в некоторый ФБ-приемник. Поскольку в расчет принимаются все возможные модели выполнения, нельзя однозначно предсказать порядок прибытия на ФБ-приемник сигналов (и их последователей), сгенерированных в некотором выполнении ФБ-источника. Поэтому надо ориентировать ФБ-приемник на прием сигналов в

различной последовательности. В качестве примера можно привести две цепочки выполнения и передач сигналов:

1) $fb_i.ee \rightarrow cAck_i.setN$;

2) $fb_i.eo_1 \rightarrow buf_j.ei_1 \rightarrow buf_j.ack_1 \rightarrow cAck_i.ack.$

Поскольку неизвестно, какой из этих сигналов ($cAck_i.setN$ или $cAck_i.ack_1$) придет первым, то блок *CountAck* был спроектирован с учетом этой ситуации. В итоге особенностью данного ФБ можно считать его невосприимчивость к порядку прибытия сигналов на событийные входы *setN* и *ack*.

Общая структура результирующей системы ФБ, построенной в соответствии с шаблоном «Циклическая модель выполнения», приведена на рис. 8.21. Входная переменная *exList* определяет линейный список выполняемых ФБ.

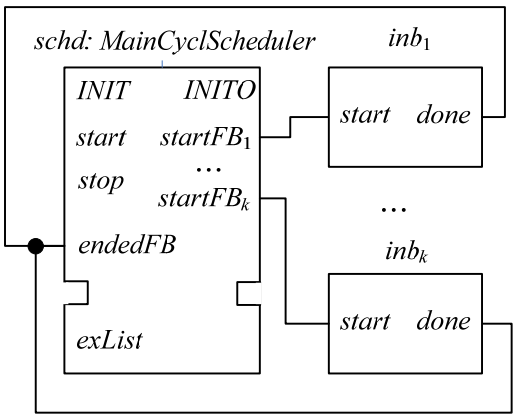


Рис. 8.21. Общая структура систем ФБ, построенных в соответствии с шаблоном «Циклическая модель выполнения»

Выполнение ФБ в предлагаемой реализации циклической модели выполнения имеет следующие особенности:

1) выполнение ФБ состоит из последовательной обработки всех сигналов из входного буфера этого ФБ с последующим запоминанием сгенерированных этим ФБ выходных сигналов в соответствующих буферах. Причем обработка входных сигналов производится в порядке их поступления на ФБ;

2) порядок обработки входных сигналов определяется режимом работы буфера. Для того, чтобы сохранить порядок обработки входных сигналов, используемых в исходной модели выполнения, необходимо выбрать соответствующий тип буфера или соответствующий режим его работы (в случае многорежимного буфера). Сигналы выводятся из буфера в ФБ последовательно, один за одним;

3) порядок передачи сигналов между ФБ регламентируется правилами, используемыми в целевой модели выполнения и режимом работы буфера. При использовании буфера *FIFO* порядок передачи сигналов будет совпадать с порядком передачи сигналов в целевой модели выполнения.

В схеме на рис. 8.19 имеются циклы (обратные связи), в том числе множественные. В общем случае событийные циклы являются опасными, они могут породить такие проблемы при выполнении, как переполнение очередей событий. Однако в данном случае система синхронизации записи сигналов, основанная на использовании ФБ *CountAck*, будет обеспечивать корректное функционирование системы.

Обобщенный алгоритм функционирования системы ФБ, построенной в соответствии с шаблоном «Циклическая модель выполнения», приведен на рис. 8.22. Алгоритм написан на псевдокоде. Под обработкой *j*-го сигнала из входного буфера *i*-го ФБ в алгоритме понимается выполнение *i*-го ФБ при наличии на его *j*-м событийном входе сигнала. Алгоритм поддерживает «хороший» сценарий функционирования системы ФБ, определенный на рис. 1.16. В алгоритме используется обозначение *L* для мощности списка *exList*.

```
-- Инициализация
Установить список выполнения ФБ exList (для диспетчера)
Проинициализировать рабочие ФБ
Проинициализировать буфера
i=1
Запустить диспетчер
-- Выполнение
ЦИКЛ ПОКА (не остановлен диспетчер)
    Выбрать очередной (i-й) ФБ из списка exList
    j=1
    ЦИКЛ ПОКА (входной буфер i-го ФБ не пуст)
        Обработать j-й сигнал из входного буфера i-го ФБ
        j=j+1
    КОНЕЦ_ЦИКЛА
    i= (i+1) mod L
КОНЕЦ_ЦИКЛА
```

Рис. 8.22. Обобщенный алгоритм функционирования системы ФБ в рамках циклической модели выполнения

На рис. 8.23 представлена диаграмма последовательности языка *UML*, описывающая поведение структуры типа *inb*. Она также частично описывает поведение общей структуры. В диаграмме последовательности использовались модификации, которые были введены в профиле *UML-FB* [115], а именно: на стрелке взаимодействия записывается два атрибута – имя выходного контакта (около начала стрелки) элемента-источника и имя входного контакта (около конца стрелки) элемента-приемника. Если оба имени совпадают, то может записываться только одно имя. Следует также отметить, что диаграмма последовательности описывает только один (хотя может и самый характерный) из множества сценариев, поэтому ее нельзя рассматривать как полное описание функционирования системы.

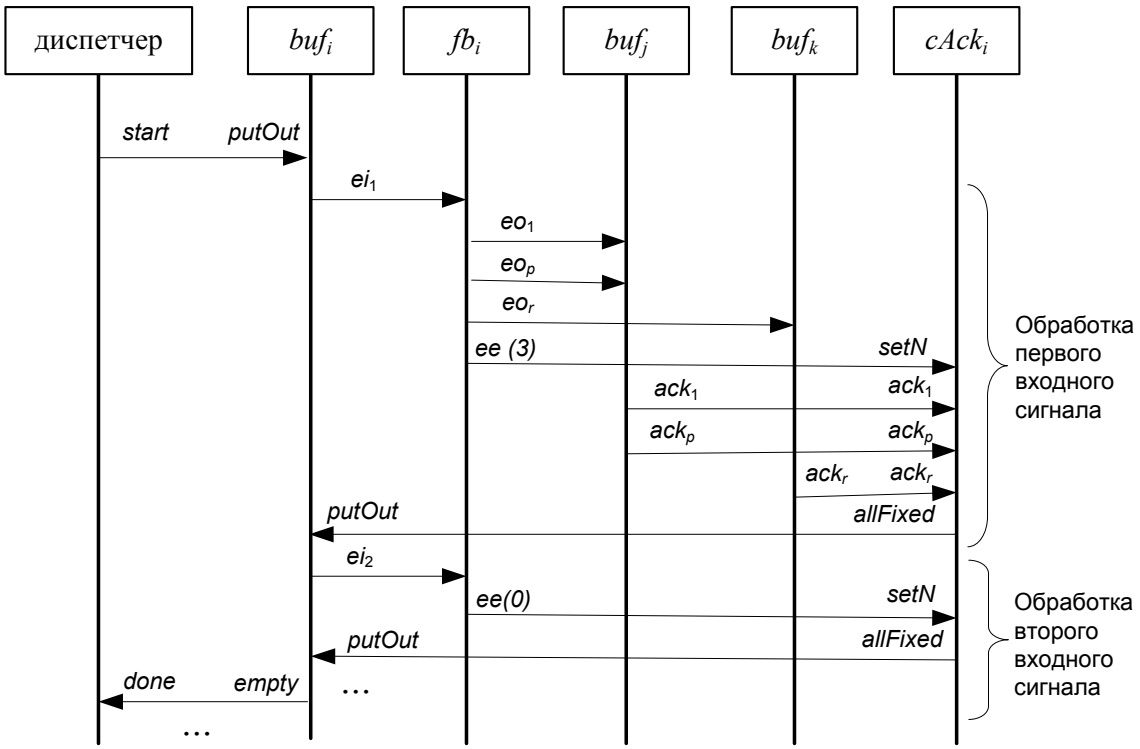


Рис. 8.23. Диаграмма последовательности,
описывающая функционирование системы типа *inb*

Более подробное описание шаблона «Циклическая модель выполнения» можно найти в [50].

8.7. Шаблон реализации «Синхронная модель выполнения»

Ниже рассматривается двухфазная синхронная модель с гранулой вычисления в виде ФБ. Схема буферизации шаблона «Синхронная модель» проиллюстрирована (на основе базовой системы ФБ из рис. 8.17) на рис. 8.24.

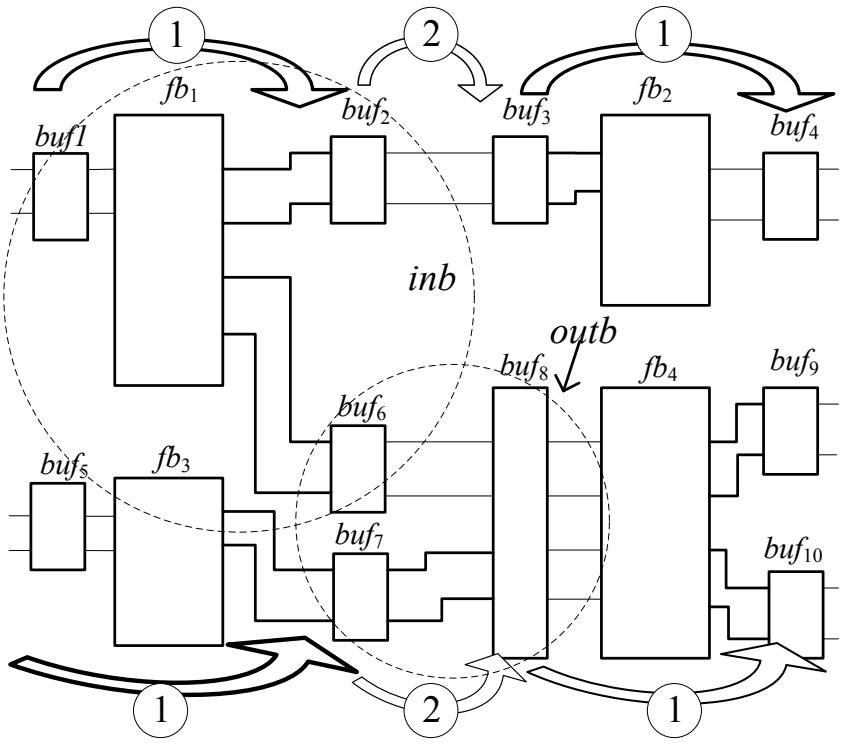


Рис. 8.24. Буферирование в системе ФБ, реализованной в рамках синхронной модели выполнения

Как видно из рис. 8.24, буферируются как входные сигналы ФБ, так и выходные. В результате этого каждый ФБ имеет один буфер входных сигналов и столько буферов выходных сигналов, сколько ФБ-последователей (по событийным связям) имеет рассматриваемый ФБ. Основные потоки событий в шаблоне «Синхронная модель выполнения» отражаются двумя структурами: 1) типа *inb* (например, buf_1, fb_1, buf_2, buf_6); 2) *outb* (buf_6, buf_7 и buf_8). В последнем случае выходными буферами являются buf_6 и buf_7, а входным буфером – buf_8.

Работа системы представляется в виде чередующихся фаз типов 1 и 2 (см. рис. 8.24). На первой фазе все события из входных буферов обрабатываются в соответствующих ФБ и после этого выходные сигналы записываются в выходные буфера. На второй фазе события из выходных буферов ФБ-источников перемещаются во входные буфера ФБ-приемников. Детализацию структуры типа *outb* можно найти в [51].

Обобщенный последовательный алгоритм функционирования систем ФБ, построенных в соответствии с принятым ШМОРСФБ «Синхронная модель выполнения», приведен на рис. 8.25. Следует отметить, что последовательный алгоритм может быть легко преобразован в параллельный путем замены последовательного оператора

ДЛЯ КАЖДОГО (*for each*) параллельным оператором ДЛЯ ВСЕХ (*for all*).

Проинициализировать рабочие ФБ -- *Инициализация*
Проинициализировать буфера
Запустить диспетчер
ЦИКЛ ПОКА (не остановлен диспетчер)
  ДЛЯ КАЖДОГО блока *fb* -- *Фаза* 1
   ДЕЛАТЬ
      Обработать в блоке *fb* каждый сигнал из входного буфера блока *fb*
      и поместить сгенерированные выходные сигналы
      в соответствующие выходные буфера
   КОНЕЦ
  ДЛЯ КАЖДОГО выходного буфера *bufOut* -- *Фаза* 2
   ДЕЛАТЬ
      Передвинуть каждое событие из выходного буфера *bufOut*
      в соответствующие входные буфера
   КОНЕЦ
КОНЕЦ_ЦИКЛА

Рис. 8.25. Обобщенный алгоритм функционирования системы ФБ в рамках синхронной двухтактной модели выполнения

Общая структура результирующей системы ФБ, построенной в соответствии с шаблоном «Синхронная модель выполнения», приведена на рис. 8.26, а диаграмма последовательности, описывающая функционирование данной системы, – на рис. 8.27.

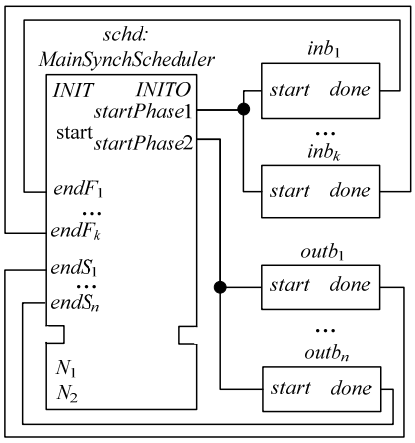


Рис. 8.26. Общая структура систем ФБ, построенных в соответствии с шаблоном «Синхронная модель выполнения»

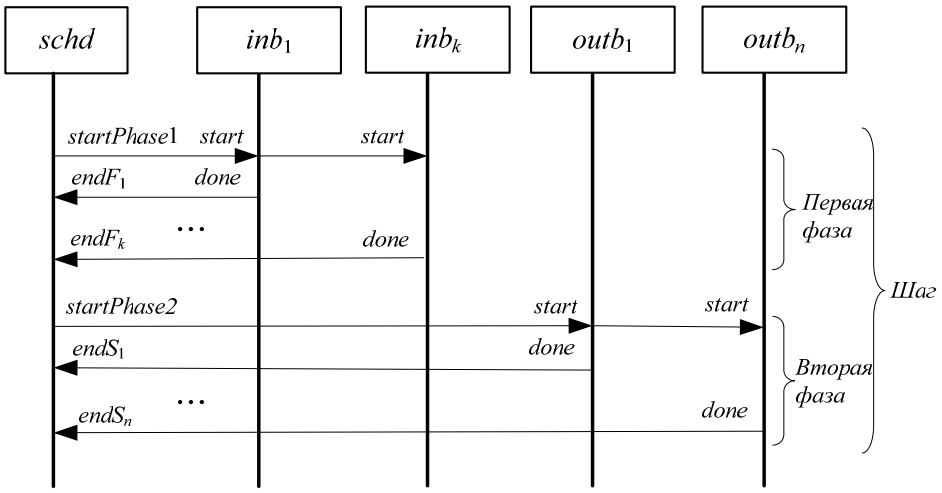


Рис. 8.27. Диаграмма последовательности, описывающая функционирование системы ФБ, построенной в соответствии с шаблоном «Синхронная модель выполнения»

Более подробное описание шаблона «Синхронная модель выполнения» можно найти в [51].

8.8. Примеры применения шаблонов

8.8.1. Система круиз-контроля в циклической модели выполнения

Ниже рассматривается использование ШМОРСФБ на примере составного ФБ, реализующего простой круиз-контроль в автомобиле. Данный ФБ является слегка модифицированной версией ФБ, представленного в [257]. Как видно из рис. 8.28, этот ФБ состоит из четырех ФБ, каждый из которых моделирует некоторый компонент системы.

Система круиз-контроля активируется рычагом, включающим в свой состав кнопки *Accel* и *Off*. Всякий раз, когда удерживается кнопка *Accel*, будет генерироваться последовательность событий *AccelHold* для пошагового ускорения автомобиля. Когда кнопка отжата, выдается событие *AccelRelease* и скорость на этот момент времени фиксируется. Требуемая скорость будет сообщаться круиз-контроллеру, который в свою очередь будет поддерживать эту скорость путем соответствующей регулировки позиции дросселя. Отдельная подсистема вычисляет текущую скорость в каждый такт системных часов и изменяет состояние круиз-контроллера. Режим круиз-контроля деактивируется при нажатии кнопки *Off*. Для краткости изложения в данном подразделе не представлены диаграммы *ECC* рассматриваемых ФБ, их можно найти в [257].

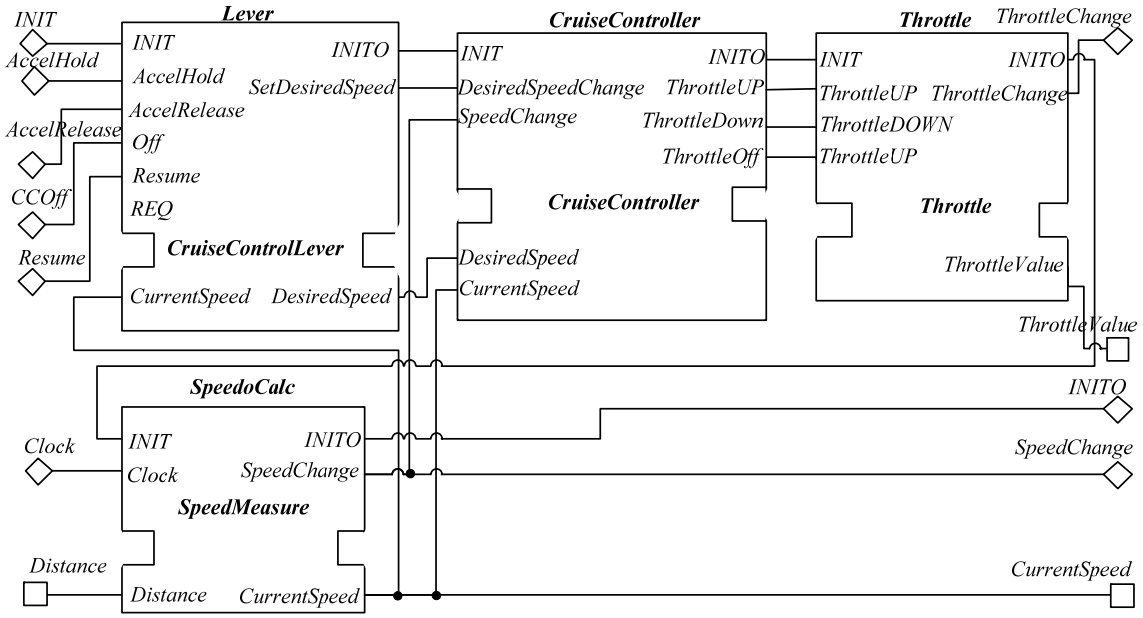


Рис. 8.28. Составной ФБ *CruiseControl*,
спроектированный для выполнения в циклической модели выполнения

Составной ФБ *CruiseControl* был спроектирован для выполнения в циклической модели. Применяя циклический шаблон, можно создать эквивалентный составной ФБ, который будет выполняться корректно в любой среде выполнения, изначально не поддерживающей циклическую семантику.

Результат применения шаблона «Циклическая модель выполнения» представлен на рис. 8.29.

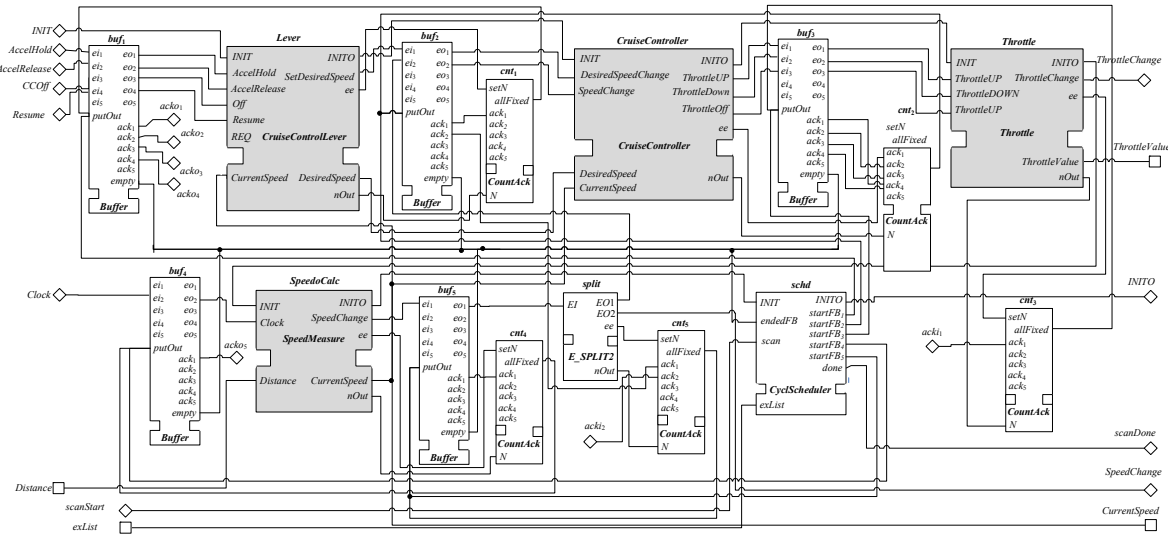


Рис. 8.29. Составной ФБ *CruiseControl*, спроектированный
в соответствии с шаблоном «Циклическая модель выполнения»

Результирующий ФБ получился более сложным, так как был применен полный шаблон проектирования для того, чтобы гарантировать эквивалентное поведение результата трансформации в любой модели выполнения, включая и те модели, где концепция «единственного прогона» не поддерживается (например, в аппаратных реализациях или модели *NPMTR*, принятой в *FBDK*). Такая «устойчивость» достигается использованием блока *CountAck* для определения момента окончания работы буфера. Для других моделей выполнения этот ФБ может быть опущен (выход *ee* рабочего ФБ будет корректно сигнализировать об окончании его работы и активировать буфер). Все это существенно упрощает результат проектирования.

В ходе трансформации в интерфейс составного ФБ были добавлены следующие сигналы:

startScan – начало сканирования компонентных ФБ в порядке, определенном в *exList*;

scanDone – сканирование завершено;

$acki_i$ – подтверждение записи во внешний буфер, расположенный за пределами этой сети ФБ;

$acko_j$ – выходной сигнал, подтверждающий запись в локальный буфер, посылаемый внешнему блоку *CountAck.*

Для корректной реализации развилки сигналов из событийного выхода *SpeedChange* блока *SpeedoCalc* был использован явный ФБ *E_SPLIT*. Этот ФБ также был подвергнут трансформации в соответствии с циклическим шаблоном. Результирующий ФБ назван *E_SPLIT2*.

Рассмотрим функционирование спроектированной сети ФБ (см. рис. 8.29) в среде выполнения *FBDK*, где для проводки сигналов используется так называемый метод «прямого вызова». При этом последовательность вызовов ФБ может быть описана алгоритмом обхода графа в глубину. В этом случае проводка сигнала будет остановлена в первом же буфере, что побуждает интерпретатор возвратиться назад к ФБ-источнику события и дать ему возможность выдать следующее событие. Так образом, последовательно, друг за другом все выходные сигналы, выданные рабочим ФБ, будут записываться в буфер, пока базисный ФБ не перейдет в терминальное *EC*-состояние. Таким образом, обход в глубину преобразуется в обход в ширину, который подходит для целевой семантики выполнения.

8.8.2. Вычисление итерационной суммы в синхронной модели выполнения

Ниже приводится иллюстрация использования шаблона «Синхронная модель выполнения» на примере системы ФБ для вычисления итерационной суммы: $result = c + \sum_{k=0}^{n} (a \cdot k)$, где a, c и n – константы. Составной ФБ, реализующий данную операцию, представлен на рис. 8.30. Данный ФБ включает следующие компонентные ФБ: умножитель, сумматор, инкрементор и компаратор.

После фазы инициализации (по сигналу *init*) приход сигнала на вход *start* приводит к вычислительным действиям, по завершении которых должен выдаваться выходной сигнал *end*, сопровождаемый выдачей результирующей суммы. Следует заметить, что выполнение данного блока в инструментальной среде *FBDK*, поддерживающей модель выполнения *NPMTR*, приводит к неправильному результату. Объяснение этого результата представлено в [50]. Данную ситуацию можно исправить, используя другие модели выполнения (в частности, циклическую или синхронную).

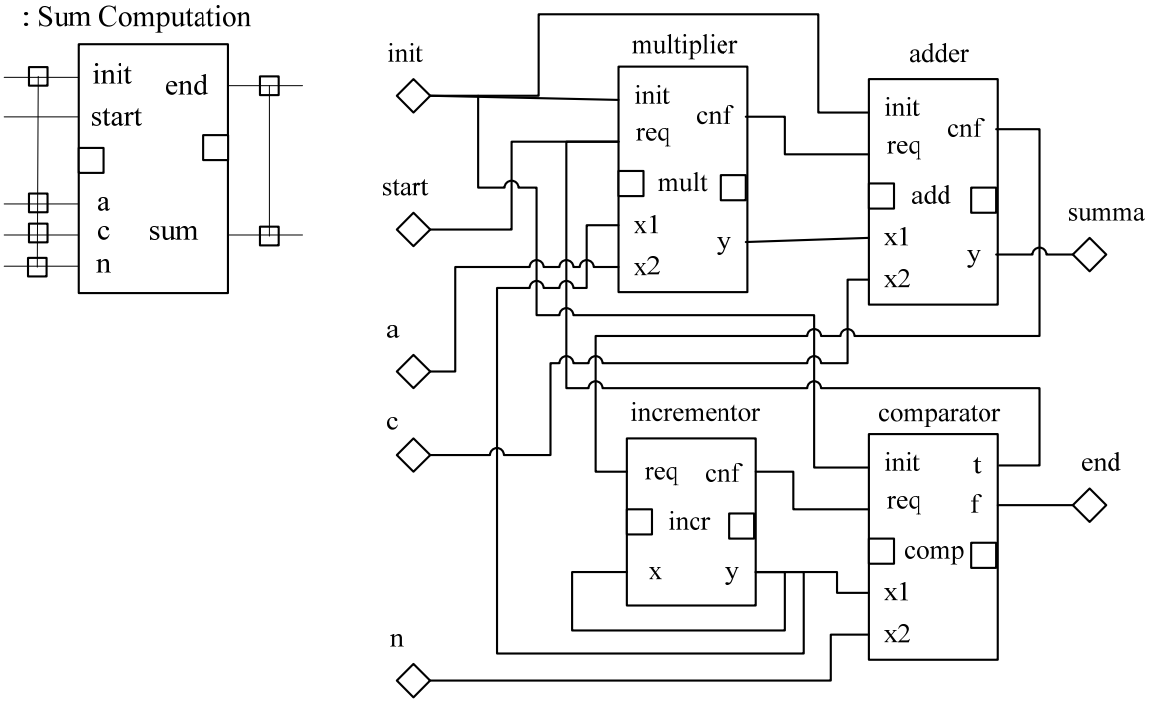


Рис. 8.30. Составной ФБ для вычисления итерационной суммы

При трансформации составного ФБ (см. рис. 8.30) в соответствии с шаблоном «Синхронная модель выполнения» в интерфейс

сгенерированного ФБ были добавлены дополнительные событийные входы для запуска и остановки диспетчера, входящего в состав ФБ.

На рис. 8.31 представлена сеть ФБ для вычисления итерационной суммы, построенная в соответствии с шаблоном «Синхронная модель выполнения». Поскольку рассматриваемый пример предназначен для иллюстративных целей в системе *FBDK*, а не для работы в составе реальной системы, то некоторые связи на рис. 8.31 не укладываются в рамки используемого шаблона. Например, не подтверждается запись во входные буфера рабочих ФБ сигналов, приходящих с событийных входов извне (от пользователя). Как можно увидеть из рис. 8.31, существуют связи от входа оболочки *init* к входам *ei*1 блоков *inBuff_mult*, *inBuff_add*, *inBuff_incr* и *inBuff_comp*, однако нет исходящих связей из событийных выходов *ack*1 этих ФБ, по которым посылались бы квитанции о записи сигнала в буфер.

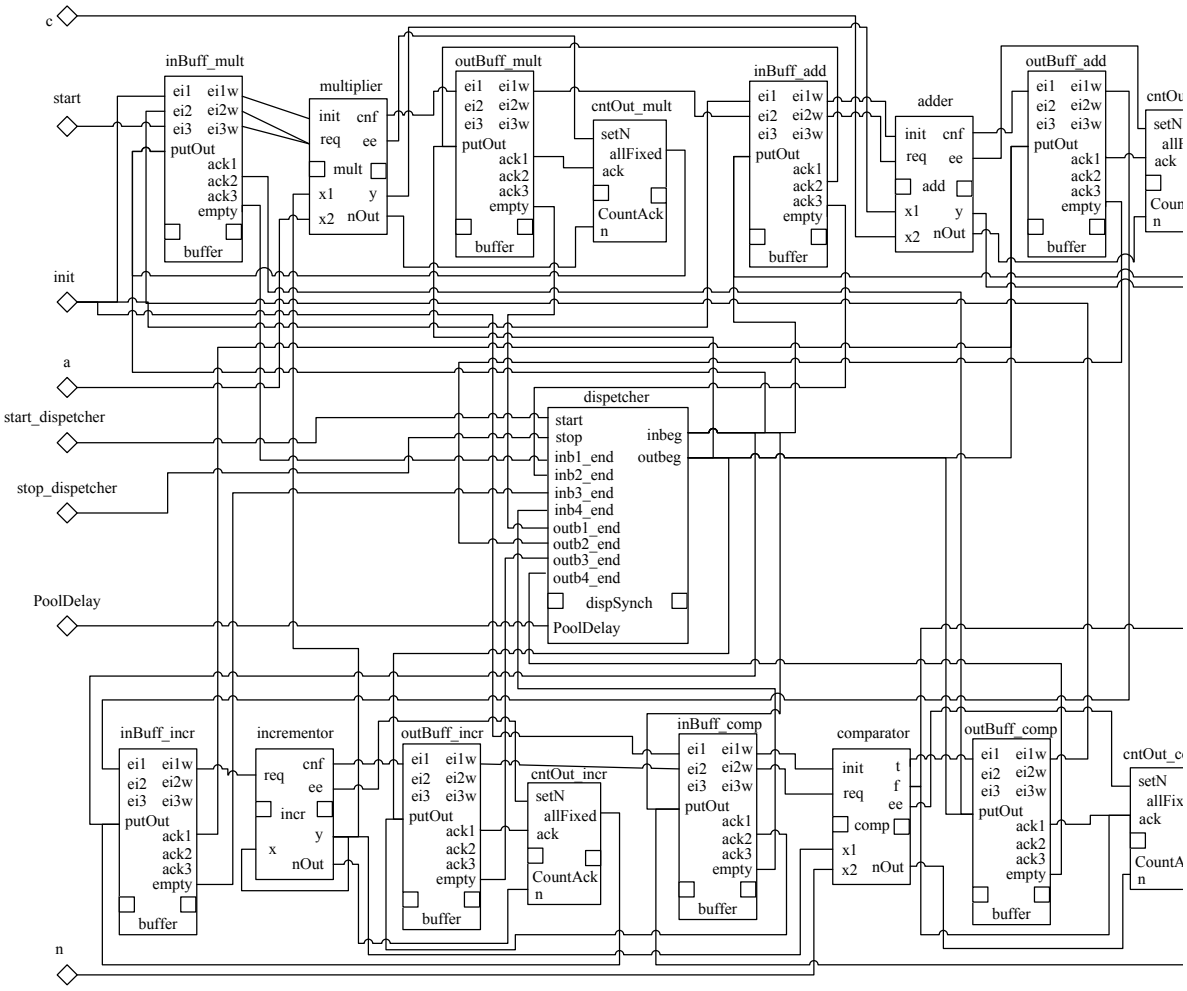


Рис. 8.31. Содержимое составного ФБ для вычисления итерационной суммы, построенного в соответствии с шаблоном реализации «Синхронная модель выполнения»

Кроме того, в представленном примере используется диспетчер, включающий блок временной задержки *E_DELAY*. Он используется для того, чтобы «разорвать» циклы, существующие в системе ФБ (они являются недопустимыми в *FBDK*). В модифицированной схеме выполнения после окончания цикла опроса и выполнения всех рабочих блоков, диспетчер берет тайм-аут, по истечении которого начинается новый цикл. Для задания времени тайм-аута используется входная переменная *PoolDelay*.

Тестирование составного ФБ из рис. 8.31, разработанного в соответствии с синхронным шаблоном, показало правильные результаты по вычислению итерационной суммы.

8.9. Оценка сложности

8.9.1. Оценка сложности накладных расходов при преобразовании составных ФБ

Сложность результирующей системы ФБ при применении ШМОРСФБ существенно зависит от принятой модели выполнения, разбиения системы ФБ на иерархические уровни и топологии сетей ФБ, входящих в систему.

Коэффициент сложности от использования ШМОРСФБ (относительно числа ФБ) будем рассчитывать по формуле

$$K = N_{\text{рез}}/N_{\text{исх}} = (N_{\text{исх}} + N_{\text{доп}})/N_{\text{исх}} = 1 + N_{\text{доп}}/N_{\text{исх}},$$

где $N_{\text{исх}}$ – число ФБ в исходной системе ФБ; $N_{\text{рез}}$ – число ФБ в результирующей системе ФБ; $N_{\text{доп}}$ – число (дополнительных) ФБ, появившихся в исходной системе ФБ в результате применения ШМОРСФБ.

Оценка сложности накладных расходов при использовании шаблона «Циклическая модель выполнения»

Оценим сложность результирующей системы ФБ при следующих ограничениях – система ФБ представляет одноуровневую плоскую связную сеть ФБ без внешних входов-выходов. Пусть M – число обобщенных событийных связей между ФБ. Будем считать, что между ФБ fb_i и fb_j существует обобщенная событийная связь, если между ними существует хотя бы одна ординарная событийная связь. Число дополнительных ФБ в системе при использовании шаблона определяется как

$$N_{\text{доп}} = N_{buf} + N_{ack},$$

где $N_{buf} = N_{\text{исх}}$ – число (входных) буферов ФБ; $N_{ack} = M$ – число счетчиков квитанций.

Следует заметить, что если обобщенная событийная связь представляет только одну ординарную событийную связь, то ее можно исключить из рассмотрения, поскольку она не требует использования счетчика квитанций.

Рассчитаем коэффициент сложности для циклического шаблона:

$$K_{\text{цикл}} = 1 + N_{\text{доп}}/N_{\text{исх}} = 1 + (N_{\text{исх}} + M)/N_{\text{исх}} = 2 + M/N_{\text{исх}}.$$

Как видно из приведенной формулы, в результирующей системе число ФБ как минимум в два раз больше числа ФБ в исходной системе.

В наихудшем случае – при топологии сети ФБ в виде полного графа

$$M = N_{\text{исх}} (N_{\text{исх}} - 1);$$

$$K'_{\text{цикл}} = 1 + N_{\text{доп}}/N_{\text{исх}} = 1 + (N_{\text{исх}} + M)/N_{\text{исх}} = 1 + N_{\text{исх}} \approx N_{\text{исх}}.$$

В этом случае коэффициент сложности растет пропорционально числу ФБ в исходной системе. Следует отметить, что на практике топология сети ФБ в виде полного графа маловероятна.

Оценка сложности накладных расходов при использовании шаблона «Синхронная модель выполнения»

Оценим сложность результирующей системы ФБ при применении синхронного шаблона при ограничениях, принятых для циклического шаблона.

Пусть L – число ФБ со степенью захода (по обобщенным событийным связям) большей или равной двум. Тогда

$$N_{\text{доп}} = N'_{buf} + N''_{buf} + N_{ack} + N_{mov},$$

где $N'_{buf} = N_{\text{исх}}$ – число входных буферов ФБ; $N_{buf} = M$ – число выходных буферов ФБ; $N_{ack} = M$ – число счетчиков квитанций; $N_{mov} = L$ – число блоков управления для перемещения сигналов из выходных буферов во входные.

$$K_{\text{синхр}} = 1 + N_{\text{доп}}/N_{\text{исх}} = 1 + (N_{\text{исх}} + 2M + L)/N_{\text{исх}} = 2 + (2M + L)/N_{\text{исх}}.$$

Как видно из приведенной формулы, в результирующей системе как минимум в два раз больше ФБ, чем в исходной системе.

Рассмотрим соотношение коэффициентов сложностей для синхронной и циклической моделей:

$$k_{21} = K_{\text{синхр}}/K_{\text{цикл}} = (2 + (2M + L)/N_{\text{исх}}) / (2 + M/N_{\text{исх}}) \approx$$
$$\approx (2M + L)/M = 2 + L/M,$$

при $M/N_{\text{исх}} >> 2$.

Как видно из приведенной формулы, сложность синхронного шаблона как минимум в два раза больше сложности циклического. (Примечание: это верно, если в исходной системе ФБ число обобщенных событийных связей значительно превышает число ФБ.)

8.9.2. Оценка сложности накладных расходов при преобразовании базисных ФБ

В данном случае сложность будем рассчитывать, принимая в учет только диаграмму *ECC*. При оценке структурной сложности, вносимой использованием шаблонов, можно ориентироваться на число *EC*-состояний, число *EC*-переходов, или их обоих вместе. В дальнейшем будем учитывать только число *EC*-состояний.

Коэффициент сложности от использования шаблона будем рассчитывать по формуле

$$K_{\text{баз}} = S_{\text{рез}}/S_{\text{исх}} = (S_{\text{исх}} + S_{\text{доп}})/S_{\text{исх}} = 1 + S_{\text{доп}}/S_{\text{исх}},$$

где $S_{\text{исх}}$ – число *EC*-состояний в исходной *ECC*; $S_{\text{рез}}$ – число *EC*-состояний в результирующей *ECC*; $S_{\text{доп}}$ – число (дополнительных) *EC*-состояний, появившихся в исходной *ECC* в результате применения шаблона.

При расчетах будем предполагать, что предварительно исходная *ECC* была подвергнута рефакторингу в соответствии с работой [246], т.е. сложность самого рефакторинга учитывать не будем.

Рассмотрим каждое правило преобразования, представленное в статье, отдельно на предмет вносимой сложности.

При использовании *Правила 3* число вновь сгенерированных *EC*-состояний будет определяться как $S'_{\text{доп}} = S_{\text{исх}} \times E$, где $S_{\text{исх}}$ – число *EC*-состояний в исходной *ECC*; E – число событийных входов ФБ.

При использовании *Правила 4* число вновь сгенерированных *EC*-состояний (в худшем случае) будет определяться как $S''_{\text{доп}} = T$, где T – число *EC*-переходов, заканчивающихся в узловых *EC*-состояниях (т.е. в тех *EC*-состояниях, в которые заходит более одной дуги).

При этом коэффициент будет рассчитываться таким образом:

$$K_{\text{баз}} = (S_{\text{исх}} + S'_{\text{доп}} + S''_{\text{доп}})/S_{\text{исх}} =$$
$$= 1 + (S'_{\text{доп}} + S''_{\text{доп}})/S_{\text{исх}} = 1 + (S_{\text{исх}} \times E + T)/S_{\text{исх}} =$$
$$= 1 + E + T/N_{\text{исх}}.$$

Если $S_{\text{исх}} >> T$, то $K_{\text{баз}} \approx 1 + E$.

Как можно заметить из формулы, коэффициент сложности (относительно базисного ФБ) растет пропорционально числу событийных входов в данном ФБ.

Если число (типов) базисных ФБ в исходной системе равно n, то коэффициент сложности для всех базисных ФБ рассчитывается следующим образом: $K_{\text{áàç}}^{all} = \sum_{i=1}^{n}(1 + E_i)$, где E_i – число событийных входов в i-м базисном ФБ.

Заключение

В данной работе заложены основы теории и технологии разработки распределенных компонентно-базированных систем управления промышленной автоматики нового поколения на основе стандарта IEC 61499. При проведении исследований использовались методы математического и машинного моделирования, обобщения, аналогии, а также передовые технологии разработки программного обеспечения. Основные результаты работы: 1) разработаны формальные модели систем функциональных блоков (ФБ) для верификации и оценки производительности; 2) разработаны модели операционной семантики ФБ; 3) предложен ряд новых моделей выполнения ФБ; 4) предложен метод семантического анализа проектов стандарта IEC 61499, основанный на онтологическом представлении систем ФБ; 5) предложен графотрансформационный подход к синтезу формальных моделей систем ФБ; 6) предложен метод рефакторинга диаграмм *ECC* базисных ФБ; 7) предложен подход к верификации систем управления на основе ФБ; 8) разработан метод обеспечения портабельности систем ФБ.

Экономическая эффективность от использования предложенных методов и средств достигается за счет ускорения процесса проектирования, повышения степени надежности проектируемых систем, а также обеспечения портабельности управляющих программ. Значимость работы определяется ответственностью управляющего программного обеспечения за безопасное функционирование различных производственных процессов, в том числе связанных с потенциальными угрозами жизни человека и окружающей среде.

Можно выделить следующие задачи в рамках дальнейших исследований по данной теме: 1) разработка методов проектирования систем управления на основе управления онтологиями; 2) аппаратная реализация реконфигурируемых систем управления с использованием языка VHDL и технологии ПЛИС с поддержкой нескольких моделей выполнения; 3) разработка систем управления для реальных секторов экономики (например, для нефтегазовой промышленности, железнодорожного транспорта и т.д.).

Библиографический список

1. Альшин, Д. В. Прямой конвертер UML-FB-диаграмм в XML-представление функциональных блоков / Д. В. Альшин, В. Н. Дубинин. – URL: http://alice.pnzgu.ru/~dvn/fb61499/uml_fb/converters/alshin/index.htm

2. Ахо, А. Теория синтаксического анализа, перевода и компиляции. / А. Ахо, Дж. Ульман. – М. : Мир, 1978. – Т.1 Синтаксический анализ. – 616 с.

3. Брауде, Э. Дж. Технология разработки программного обеспечения / Э. Дж. Брауде. – СПб. : Питер, 2004. – 655 с.

4. Бусленко, В. Н. Автоматизация имитационного моделирования сложных систем / В. Н. Бусленко. – М. : Наука, 1977. – 240 с.

5. Бусленко, Н. П. Моделирование сложных систем / Н. П. Бусленко. – М. : Наука, 1968. – 356 с.

6. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч, Р. А. Максимчук, М. У. Энгл, Б. Дж. Янг, Дж. Коналлен, К. А. Хьюстон. – 3-е изд. – М. : Вильямс, 2008. – 721 с.

7. Буч, Г. Язык UML. Руководство пользователя : пер. с англ. / Г. Буч, Дж. Рамбо, А. Джекобсон. – М. : ДМК Пресс; СПб. : Питер, 2004. – 432 с.

8. Вашкевич, Н. П. Вопросы разработки операционной семантики функциональных блоков IEC 61499 / Н. П. Вашкевич, В. Н. Дубинин // Программные системы и вычислительные методы. – 2012. – № 1. – С. 10–16.

9. Вашкевич, Н. П. Формализованное описание последовательной модели выполнения функциональных блоков / Н. П. Вашкевич, В. Н. Дубинин, В. В. Вяткин // Вычислительные системы и технологии обработки информации : межвуз. сб. науч. тр. – Вып. 10 (33). – 2011. – С. 45–61.

10. Вельдер, С. Э. Верификация автоматных программ / С. Э. Вельдер, М. А. Лукин, А. А. Шалыто, Б. Р. Яминов. – СПбГУ ИТМО, 2011. – 242 с.

11. Воеводин, В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – СПб. : БХВ-Петербург, 2002. – 608 с.

12. Гаврилова, Т. А. Базы знаний интеллектуальных систем / Т. А. Гаврилова, В.Ф. Хорошевский. – СПб. : Питер, 2000. – 384 с.

13. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб. : Питер, 2001. – 368 с.

14. Гома, Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений / Х. Гома. – М. : ДМК, 2002. – 704 с.

15. Грейвс, М. Проектирование баз данных на основе XML : пер. с англ. / М. Грейвс. – М. : Вильямс, 2002. – 640 с.

16. Дубинин, В. Н. Анализ расширенных NCES-сетей на основе метода Model Checking / В. Н. Дубинин, Х.-М. Ханиш, В. В. Вяткин, С. К. Шестаков // Новые информационные технологии и системы (НИТиС'2010) : тр. IX Междунар. науч.-техн. конф. – Пенза, 2010. – Ч. 2. – С. 20–48.

17. Дубинин, В. Н. Асинхронное моделирование NCES-сетей / В. Н. Дубинин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2009. – № 2. – С. 3–14.

18. Дубинин, В. Н. Верификация приложений IEC 61499 на основе метода Model Checking / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2011. – № 3. – С. 44–55.

19. Дубинин, В. Н. Графо-трансформационный подход к синтезу формальных моделей систем функциональных блоков IEC 61499 / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2008. – № 4. – С. 16–26.

20. Дубинин, В. Н. Графовые трансформации для синтеза формальных моделей управляющих приложений IEC 61499 / В. Н. Дубинин // Современные информационные технологии в науке, образовании и практике : материалы VII Всерос. науч.-практ. конф. (с международным участием). – Оренбург : ОГУ, 2008. – С. 28–41.

21. Дубинин, В. Н. Изучение XML-технологий на примерах систем промышленной автоматики / В. Н. Дубинин, Л. П. Климкина // Университетское образование (МКУО-2006) : сб. ст. X Междунар. науч.-метод. конф. – Пенза, 2006. – С. 442–444.

22. Дубинин, В. Н. Использование Web-онтологий в проектировании реконфигурируемых систем промышленной автоматики / В. Н. Дубинин, В. В. Вяткин // Интеллектуальные системы (AIS'07) : тр. междунар. науч.-техн. конф. (Дивноморское). – М. : Физматлит, 2007. – Т. 2. – С. 14–22.

23. Дубинин, В. Н. Концептуальное моделирование систем управления на основе функциональных блоков IEC 61499 / В. Н. Дубинин // Вестник ТГТУ. – 2009. – № 3. – С. 467–477.

24. Дубинин, В. Н. Метамоделирование функциональных блоков IEC 61499 и NCES-сетей / В. Н. Дубинин, В. В. Вяткин // Современные информационные технологии : тр. междунар. науч.-техн. конф. – Пенза, 2008. – Вып. 8. – С. 162–169.

25. Дубинин, В. Н. Модели последовательного выполнения функциональных блоков IEC 61499 на основе динамически изменяемых приоритетов / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2007. – № 1. – С. 13–22.

26. Дубинин, В. Н. Моделирование и верификация функциональных блоков IEC 61499 с использованием SMV / В. Н. Дубинин, В. В. Вяткин // Информационные ресурсы и системы в экономике, науке и образовании : сб. ст. междунар. науч.-практ. конф. – Пенза : Приволжский Дом знаний, 2011. – С. 5–10.

27. Дубинин, В. Н. Моделирование систем транспортировки деталей на основе функциональных блоков стандарта IEC 61499 / В. Н. Дубинин // Новые информационные технологии и системы (НИТС’2002) : материалы V Междунар. науч.-техн. конф. – Пенза, 2002. – С. 32.

28. Дубинин, В. Н. Моделирование систем функциональных блоков IEC 61499 с помощью модульных арифметических NCES-сетей / В. Н. Дубинин // Новые информационные технологии и системы (НИТиС’2008) : тр. VIII Междунар. науч.-техн. конф. – Пенза, 2008. – Ч. 2. – С. 47–64.

29. Дубинин, В. Н. Моделирование функциональных блоков в системе CPN Tools / В. Н. Дубинин, В. Б. Механов, Е. К. Таранцев // Вычислительные системы и технологии обработки информации : межвуз. сб. науч. тр. – 2011. – Вып.10 (33). – С. 7–19.

30. Дубинин, В. Н. Обнаружение циклов в системах функциональных блоков IEC 61499 с использований онтологий / В. Н. Дубинин, В. В. Вяткин // Современные информационные технологии : тр. междунар. науч.-техн. конф. – Пенза, 2011. – Вып.13. – С. 149–161.

31. Дубинин, В. Н. Одноуровневое представление систем функциональных блоков / В. Н. Дубинин // Современные информационные технологии-2005 : сб. ст. междунар. науч.-техн. конф. – Пенза, 2005. – С. 59–63.

32. Дубинин, В. Н. Онтология функциональных блоков стандарта IEC 61499 / В. Н. Дубинин, В. В. Вяткин // Современные инфор-

мационные технологии : тр. междунар. науч.-техн. конф. – Пенза, 2010. – Вып. 12. – С. 113–126.

33. Дубинин, В. Н. Операционная семантика синхронных функциональных блоков IEC 61499 на основе машин абстрактных состояний. Ч. 1. Модель диспетчеров / В. Н. Дубинин, В. В. Вяткин // XXI век: итоги прошлого и проблемы настоящего Плюс. – 2012. – Вып. 4. – С. 233–240.

34. Дубинин, В. Н. Операционная семантика синхронных функциональных блоков IEC 61499 на основе машин абстрактных состояний. Ч. 2. Модели блоков и представление в SMV / В. Н. Дубинин, В. В. Вяткин // Современные информационные технологии : тр. междунар. науч.-техн. конф. – Пенза, 2011. – Вып.14. – С. 94–100.

35. Дубинин, В. Н. Построение поисково-трансформационных систем для поддержки проектирования компонентно-базированных систем промышленной автоматики / В. Н. Дубинин, В. В. Вяткин // Интеллектуальные системы (AIS’05)» и «Интеллектуальные САПР (CAD-2005) : тр. междунар. науч.-техн. конф. В 3 т. – М. : Физматлит, 2005. – Т. 2. – С. 30–35.

36. Дубинин, В. Н. Проектирование распределенных систем управления промышленными процессами с использованием UML-FB / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2004. – № 2 (11). – С. 136–146.

37. Дубинин, В. Н. Разработка визуальных имитационных моделей производственных систем на основе UML и функциональных блоков / В. Н. Дубинин, А. А. Дубравин, В. В. Черемушкин, В. В. Вяткин // Новые информационные технологии и системы (НИТС’2004) : тр. VI Междунар. науч.-техн. конф. – Пенза, Россия, 17–19 июня 2004. – Ч. 2. – С. 68–76.

38. Дубинин, В. Н. Разработка интегрированной параметризованной модели выполнения функциональных блоков IEC 61499 с использованием морфологических методов / В. Н. Дубинин, В. В. Вяткин // Современные информационные технологии-2008 : сб. ст. междунар. науч.-техн. конф. – Пенза, 2008. – Вып. 7. – С. 146–154.

39. Дубинин, В. Н. Распределенная реализация интерпретированных сетей Петри в архитектуре IEC 61499 / В. Н. Дубинин // Вычислительные системы и технологии обработки информации : межвуз. сб. науч. трудов. – Пенза : Изд-во Пенз. гос. ун-та, 2005. – Вып. 3(29). – С. 58–64.

40. Дубинин, В. Н. Реализация некоторых видов взаимодействий в системах, основанных на стандарте IEC 61499 / В. Н. Дубинин // Вычислительные системы и технологии обработки информации : межвуз. сб. науч. трудов. – Пенза : Изд-во Пенз. гос. ун-та, 2005. – Вып. 3(29). – С. 65–71.

41. Дубинин, В. Н. Рефакторинг диаграмм управления выполнением стандарта IEC 61499 / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2008. – № 2. – С.16–25.

42. Дубинин, В. Н. Семантический анализ описаний систем управления промышленными процессами на основе стандарта IEC 61499 с использованием онтологий / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2010. – № 3 (15). – С. 3–15.

43. Дубинин, В. Н. Семантическое моделирование систем промышленной автоматики / В. Н. Дубинин, Л. П. Климкина // Университетское образование (МКУО-2007) : сб. ст. XI Междунар. науч.-метод. конф. – Пенза, 2007. – С. 194–196.

44. Дубинин, В. Н. Система имитационного моделирования распределенных систем промышленной автоматики на основе функциональных блоков / В. Н. Дубинин // Новые информационные технологии и системы (НИТиС’2010) : тр. IX Междунар. науч.-техн. конф. – Пенза, 2010. – Ч. 2.– С. 9–20.

45. Дубинин, В. Н. Создание информационных систем для поддержки проектирования на основе функциональных блоков IEC 61499 / В. Н. Дубинин // Новые информационные технологии и системы (НИТС’2002) : материалы V Междунар. науч.-техн. конф. – Пенза, 2002. – С. 34–37.

46. Дубинин, В. Н. Сравнительный анализ некоторых аспектов выполнения функциональных блоков IEC 61499 / В. Н. Дубинин // Современные информационные технологии в науке, образовании и практике : материалы VII Всерос. науч.-практ. конф. (с международным участием). – Оренбург : ОГУ, 2009. – С. 19–28.

47. Дубинин, В. Н. Формализация моделей выполнения функциональных блоков IEC 61499 / В. Н. Дубинин, В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2011. – № 1. – С. 12–23.

48. Дубинин, В. Н. Формализованное описание и моделирование систем функциональных блоков IEC 61499 / В. Н. Дубинин,

В. В. Вяткин // Известия высших учебных заведений. Поволжский регион. Технические науки. – 2005. – № 5 (20). – С. 76–89.

49. Дубинин, В. Н. Формализованное описание и разработка моделей выполнения систем интерфейсов функциональных блоков IEC 61499 / В. Н. Дубинин // Современные информационные технологии–2008 : тр. междунар. науч.-техн. конф. – Пенза, 2008. – Вып.7 – С. 155–164.

50. Дубинин, В. Н. Функционально-блочная реализация моделей выполнения в системах функциональных блоков IEC 61499 / В. Н. Дубинин, В. В. Вяткин // Современные информационные технологии : тр. междунар. науч.-техн. конф. – Пенза, 2010. – Вып. 11. – С. 123–138.

51. Дубинин, В. Н. Шаблон реализации функциональных блоков «Синхронная модель выполнения» / В. Н. Дубинин, В. В. Вяткин // Современные информационные технологии : тр. междунар. науч.-техн. конф. – Пенза, 2010. – Вып. 12. – С. 102–113.

52. Замулин, А. В. Формальная модель Java-программы, основанная на машинах абстрактных состояний / А. В. Замулин // Программирование. – 2003. – № 3. – С. 15–28.

53. Карпов, Ю. Г. Model Checking. Верификация параллельных и распределенных программных систем / Ю. Г. Карпов. – СПб. : БХВ-Петербург, 2010. – 552 с.

54. Касьянов, В. Н. Графы в программировании: обработка, визуализация и применение / В. Н. Касьянов, В. А. Евстигнеев. – СПб. : БХВ-Петербург, 2003. – 1104 с.

55. Кларк, Э. Верификация моделей программ: Model Checking / Э. Кларк, О. Грамберг, Д. Пелед. – М. : МЦНМО, 2002. – 416 с.

56. Ковалёв, С. П. Применение формальных методов для обеспечения качества вычислительных систем / С. П. Ковалёв // Вестник Новосибирского государственного университета. Математика, механика, информатика. – 2004. – Т. IV. – Вып. 2. – С. 49–74.

57. Колесин, А. В. Система преобразования функциональных блоков в сетевые модели / А. В. Колесин, В. Н. Дубинин. – URL: http://sites.google.com/site/fb2nces/Home.

58. Кораблин, Ю. П. Семантика языков программирования: учеб. пособие / Ю. П. Кораблин. – М. : Изд-во МЭИ, 1992. – 100 с.

59. Котов, В. Е. Сети Петри / В. Е. Котов. – М. : Наука, 1984. – 158 с.

60. Кулямин, В. В. Методы верификации программного обеспечения / В. В. Кулямин // Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению «Информационно-телекоммуникационные системы», 2008. – 117 с. – URL: – http://www.ict.edu.ru/lib/index.php?id_res=5645

61. Льюис, Ф. Теоретические основы проектирования компиляторов / Ф. Льюис, Д. Розенкранц, Р. Стирнз. – М. : Мир, 1979. – 654 с.

62. Люггер, Дж. Ф. Искусственный интеллект: стратегии и методы решения сложных проблем : пер. с англ. / Дж. Ф. Люгер – М. : Вильямс, 2003. – 864 с.

63. Миронов, А. М. Верификация программ методом Model Checking / А. М. Миронов. – URL: http://intsys.msu.ru/staff/mironov/modelchk.pdf

64. Непомнящий, В. А. Прикладные методы верификации программ / В. А. Непомнящий, О. М. Рякин ; под ред. А. П. Ершова. – М. : Радио и связь. 1988. – 256 с.

65. Питерсон, Дж. Теория сетей Петри и моделирование систем / Дж. Питерсон. – М. : Мир, 1984. – 263.

66. Поликарпова, Н. И. Автоматное программирование / Н. И. Поликарпова, А. А. Шалыто. – СПб. : Питер, 2011. – 176 с.

67. Раскин, С. И. Обратный конвертер XML-представления функциональных блоков в UML-FB-диаграммы / С. И. Раскин, В. Н. Дубинин. – URL: http://alice.pnzgu.ru/~dvn/fb61499/uml_fb/converters/raskin/index.htm

68. Назаретов, В. М. Робототехнические комплексы. Искусственный интеллект. Робототехника и гибкие автоматизированные производства : в 9 кн. / В. М. Назаретов, Д. П. Ким ; под ред. И. М. Макарова. – М. : Высш. шк., 1986. – Кн. 6. Техническая имитация интеллекта : учеб. пособие для втузов. – 144 с.

69. Скопинцев, Р. Г. Подход к реализации систем функциональных блоков IEC 61499 на языке VHDL / Р. Г. Скопинцев, В. Н. Дубинин // Новые информационные технологии и системы (НИТиС’2012) : тр. X Междунар. науч.-техн. конф. – Пенза, 2012. – С. 59–64.

70. Холзнер, С. XSLT. Библиотека программиста / С. Холзнер. – СПб. : Питер, 2002. – 544 с.

71. Чамберлин, Д. W3C XML: XQuery от экспертов. Руководство по языку запросов / Д. Чамберлин, Д. Дрейпер, М. Фернандес, М. Кей, Дж. Роби, М. Рис, Ж. Симеон, Дж. Тивай, Ф. Уодлер : пер. с англ. – М. : КУДИЦ-ОБРАЗ, 2005. – 480 с.

72. Черемушкин, В. В. Визуальная имитационная модель производственной установки FESTO на основе функциональных блоков IEC 61499 [Электронный ресурс] / В. В. Черемушкин, В. Н. Дубинин. – URL: http://alice.pnzgu.ru/~dvn/fb61499/festo/visual_simulation_fbdk/index.html

73. 4DIAC – Framework for Distributed Industrial Automation. – URL: http://www.fordiac.org, 2010

74. Abstract State Machine Web-site. – URL: http://www.eecs.umich.edu/gasm/

75. AGG Web-сайт. – URL: http://tfs.cs.tu-berlin.de/agg

76. Akesson, K. Supremica – an integrated environment for verification, synthesis and simulation of discrete event systems / K. Akesson, M. Fabian, H. Flordal, R. Malik // Proceedings of the 8th International Workshop on Discrete Event Systems, WODES '06. – 2006. – P. 384–395.

77. Albers, K. Hierarchical event streams and event dependency graphs: a new computational model for embedded real-time systems / K. Albers, F. Bodmann, F. Slomka // 18th Euromicro Conference on Real-Time Systems. – 2006. – 10 p.

78. Allen, L.V. Closed-Loop Determinism for Non-Deterministic Environments: Verification for IEC 61499 Logic Controllers / L.V. Allen, K.M. Goh, D.M. Tilbury // IEEE International Conference on Automation Science and Engineering. – 2009. – P. 1–6.

79. Allen, L. V. Verication and Anomaly Detection for Event-Based Control of Manufacturing Systems. – PhD. Theses : University of Michigan, 2010. – 170 p.

80. Arakawa, N. Semantic Analysis based on Ontologies with Semantic Web Standards / N. Arakawa // Int. Conf. Computer-aided Acquisition of Semantic Knowledge (CASK-2008). – Paris : Sorbonne, 2008. – 8 p.

81. Arnold, A. Finite transition systems: semantics of communicating systems / A. Arnold. – Masson, 1994. – 177 p.

82. Automation Objects for Industrial-Process Measurement and Control Systems. Working draft of International Standard IEC // International Electrotechnical Commission (Technical Committee No. 65), 2002. – 14 p.

83. Börger, E. Abstract State Machines. A Method for High-Level System Design and Analysis / E. Börger, R. F. Staerk. – Berlin; Heidelberg : Springer Verlag, 2003. – 438 c.

84. Börger, E. Abstract State Machines: a unifying view of models of computation and of system design frameworks / E. Börger // Annals of Pure and Applied Logic. – 2005. – Vol. 133, № 1–3. – P. 149–171.

85. Börger, E. A High-Level Modular Definition of the Semantics of C# / E. Börger, N. Fruha, V. Gervasi, R. Staerk // Theoretical Computer Science. – 2005. – Vol. 336, Issue 2–3. – P. 1–35.

86. Bonfè, M. An application of software design methods to manufacturing systems supervision and control / M. Bonfè, C. Donati, C. Fantuzzi // IEEE Conf. on Control Application (IEEE-CCA). – Scotland, Glasgow, 2002. – Vol. 2. – P. 850–855.

87. Bonfè, M. Design and verification of mechatronic object-oriented models for industrial control systems / M. Bonfè, C. Fantuzzi // IEEE Conference Emerging Technologies and Factory Automation (ETFA '03), 2003. – Vol. 2. – P. 253–260.

88. Bräuer, M. An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning / M. Bräuer, H. Lochmann // The Semantic Web: Research and Applications. LNCS. – Berlin; Heidelberg : Springer Verlag, 2008. – Vol. 5021. – P. 34–48.

89. Bräuer, M. Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations / M. Bräuer, H. Lochmann // 4th International Workshop on (Software) Language Engineering (ATEM 2007) co-located with MoDELS, 2007. – 15 p.

90. Brennan, R. W. Automation objects: enabling embedded intelligence in real-time mechatronic systems / R.W. Brennan, L. Ferrarini, J. L. M. Lastra, V. Vyatkin // International Journal of Manufacturing Research. – 2006. – Vol. 1. – P. 379–381.

91. Burch, J.R. Symbolic model checking: 10^{20} states and beyond / J. R. Burch, E. M. Clarke, K. McMillan, D. Dill, L. Hwang // Proc. Fifth Annual IEEE Symposium on Logic in Computer Science. – 1990. – P. 428–439.

92. 89. Burch, J.R. Symbolic model checking with partitioned transition relations / J. R. Burch, E. M. Clarke, D. E. Long // International Conference on Very Large Scale Integration. – Scotland, Edinburgh, 1991. – P. 49–58.

93. Cadence SMV. – URL: http://www.kenmcmil.com/smv.html

94. Campos, J. C. Pattern-based Analysis of Automated Production Systems / J. C. Campos, J. Machado // 13th IFAC Symposium on Information Control Problems in Manufacturing. – Moscow, 2009. – P. 976–981.

95. Cavarra, A. Mapping UML into Abstract State Mashines: A Framework to Simulate UML Models / A. Cavarra, E. Riccobene, P. Scandurra // Studia Informatica Universalis. – 2004. – P. 367–398.

96. Čengic, G. A Control Software Development Method Using IEC 61499 Function Blocks, Simulation and Formal Verification / G. Čengic, K. Akesson // 17th IFAC World Congress, Korea, Seoul, 2008. – P. 22–27.

97. Čengić, G. Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime / G. Čengić, O. Ljungkrantz, K. Åkesson // Proc. 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2006), September 20–22, 2006. – Czech Republic, Prague, 2006. – P. 1269–1276.

98. Čengic, G. On Formal Analysis of IEC 61499 Applications, Part A: Modeling / G. Čengić, K. Åkesson // IEEE Transactions on Industrial Informatics. – 2010. – Vol. 6, № 2. – P. 136–144.

99. Čengic, G. On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics / G. Čengić, K. Åkesson // IEEE Transactions on Industrial Informatics. – 2010. – Vol. 6, № 2. – P. 145–154.

100. Chilkat XML C / C++ Library. – URL: http://www.chilkatsoft.com/xml-library.asp

101. Christensen, J. H. Design patterns for systems engineering with IEC 61499 / J. H. Christensen // Fachtagung «Verteilte Automatisierung – Modelle und Methoden für Entwurf, Verifikation, Engineering und Instrumentierung». – Magdeburg : Otto-von-Guericke-Universität, 2000. – P. 63–71.

102. Christensen, H. J. HMS/FB Architecture and its Implementation / H. J. Christensen // Agent-Based Manufacturing: Advances in the Holonic Approach. – Berlin/Heidelberg : Springer-Verlag, 2003. – P. 53–87.

103. Christensen, H. J. IEC 61499 Architecture, Engineering Methodologies and Software Tools / H. J. Christensen // Knowledge and Technology Integration in Production and Services. – Boston/Dordrecht/ London : Kluwer Academic Publishers, 2002. – P. 221–236.

104. Cimatti, A. NuSMV 2: An OpenSource Tool for Symbolic Model Checking / A. Cimatti, E. D. Clarke, E. Giunchiglia, F. Giunchiglia, P. Marco, R. Marco, R. Sebastiani, A. Armando. – Technical Report № DIT-02-0016, Computer Science Department, Carnegie Mellon University, 2002. – 9 P. – URL: http://repository.cmu.edu/compsci/430

105. Cladera, R. A schema for the improved allocation of functionality in distributed industrial process-measurement and control systems /

R. Cladera, G. Frey // XXVIII.ASR'2003 Seminar, Instruments and Control, Ostrava, 2003. – 6 p.

106. Clarke, E. M. Compositional model checking / E. M. Clarke, D. Long, K. McMillan // Proceedings of the Fourth Annual Symposium on Logic in computer science, IEEE Press Piscataway, NJ, USA, 1989. – P. 353–362.

107. Clarke, E. M. Modular translation of statecharts to SMV / E. M. Clarke, W. Heinle. – Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2000. – 40 p.

108. Complexity of reasoning in Description Logics. – URL: http:// www.cs.man.ac.uk/~ezolin/dl/

109. Compton, K. An Automatic Verification Tool for UML / K. Compton, Y. Gurevich, J. Huggins, W. Shen. – Technique Report CSE-TR-423-00, Dept. of EECS, The University of Michigan, 2000. – 49 c.

110. Corbett, J. C. Evaluating Deadlock Detection Methods for Concurrent Software / J. C. Corbett // IEEE Transactions on Software Engineering. – Vol. 33, № 3. – 1996. – P. 161–180.

111. Dai, W. IEC 61499 Ontology Model for Semantic Analysis and Code Generation / W. Dai, V. Dubinin, V. Vyatkin // IEEE 9th International Conference on Industrial Informatics (INDIN'2011), Lisbon, Portugal, 2011. – P. 597–602.

112. Dai, W. Ontology-based Design Recovery and Migration between IEC 61499 – compliant Tools / W. Dai, V. Vyatkin, V. Dubinin // 37th Annual Conference of the IEEE Industrial Electronics Society(IECON 2011). – Melbourne, Australia, 2011. – P. 4332–4337.

113. Dimitrova, D. Formal approach for modeling and verification of IEC 61499 function blocks / D. Dimitrova, G. Frey, I. Bachkova // Int. Conf. Advanced Manufacturing Technologies (AMTECH 2005). – Russe, Bulgaria. – Vol. 44.– Book 2, 2005. – P. 731–736.

114. Distributed Automation Testbed. – URL: http://at.iw.uni-halle.de/~testbeds/plant.htm.

115. Dubinin, V. Engineering of Validatable Automation Systems Based on an Extension of UML Combined With Function Blocks of IEC 61499 / V. Dubinin, V. Vyatkin, T. Pfeiffer // IEEE International Conference on Robotics and Automation (ICRA'05). – Barcelona, Spain, 2005. – P. 4007–4012.

116. Dubinin, V.N. Implementation of some kinds of interactions and communications in IEC 61499 architecture / V. Dubinin // Новые

информационные технологии и системы (НИТС'2004) : тр. VI Междунар. науч.-техн. конф. – Пенза, 2004. – Ч. 2. – С. 84–92.

117. Dubinin, V. On Definition of a Formal Semantic Model for IEC 61499 Function Blocks / V. Dubinin, V. Vyatkin // EURASIP Journal on Embedded Systems. – 2008. – 10 p.

118. Dubinin, V. Refactoring of Execution Control Charts in Basic Function Blocks of the IEC 61499 Standard / V. Dubinin, V. Vyatkin // Proc. 13th IFAC Symposium on Information Control Problems in Manufacturing. – Moscow, 2009. – P. 193–198.

119. Dubinin, V. Semantics-Robust Design Patterns for IEC 61499 / V. Dubinin, V. Vyatkin // IEEE Transactions on Industrial Informatics. – 2012. – Vol. 8, Issue 2. – P. 279–290.

120. Dubinin, V. Towards a Formal Semantics of IEC 61499 Function Blocks / V. Dubinin, V. Vyatkin // 4th IEEE International Conference on Industrial Informatics (INDIN'2006). – Singapore, 2006. – P. 6–11.

121. Dubinin, V. UML-FB – a language for modeling and implementation of industrial-process measurement and control systems on the basis of IEC 61499 standard / V. Dubinin, V. Vyatkin // Новые информационные технологии и системы(НИТС'2004) : тр. VI Междунар. науч.-техн. конф.– Пенза, 2004. – Ч. 2. – С. 77–83.

122. Dubinin, V. Using Prolog for Modelling And Verification of IEC 61499 Function Blocks and Applications / V. Dubinin, V. Vyatkin, H.-M. Hanisch // 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2006), Proceedings, Prague, Czech Republic, 2006. – P. 774–781.

123. Ehrig, H. Fundamental theory for typed attributed graph transformation / H. Ehrig, U. Prange, G. Taenzer // Graph Transformation: 2nd Int. Conf. (ICGT 2004). Lecture Notes in Computer Science. – Berlin; Heidelberg : Springer Verlag, 2004. – Vol. 3256. – P. 161–177.

124. Ehrig, H. Termination criteria for model transformation / H. Ehrig, K. Ehrig, J. Lara, G. Taentzer, D. Varro, S. Varro-Gyapay // Lecture Notes in Computer Science. – 2005. – Vol. 3442. – P. 49–63.

125. Enste, U. Technical Application of Hybrid Modeling Methods to specify Function Block Systems / U. Enste, U. Epple // Automatisieungstechnik, 48. – R.Oldenburg Verlag, 2000. – 8 p.

126. Eshuis, H. A Formal Semantics for UML Activity Diagrams – Formalising Workflow Models / H. Eshuis, R. J. Wieringa. – Technical Report TR-CTIT-01-04, Centre for Telematics and Information Technology University of Twente, Enschede, 2001. – 44 p.

127. eXist-db Open Source Native XML Database. – URL: http://exist-db.org/

128. Extensible Markup Language (XML). – URL: http://www.w3.org/XML/

129. Faure, J. M. Towards IEC 61499 function blocks diagrams verification / J. M. Faure, J. J. Lesage, C. Schnakenbourg // IEEE Int. Conference on Systems, Man and Cybernetics (SMC02). – Hammamet, Tunisia, 2002. – Vol. 3.

130. FBench – Open Source Function Block Engineering Tool. – URL: http://oooneida-fbench.sourceforge.net/, 2008.

131. Feige, U. Exact analysis of hot-potato routing / U. Feige, P. Raghavan // 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, 1992. – P. 553–562.

132. Ferrarini, L. Implementation approaches for the execution model of IEC 61499 applications / L. Ferrarini, C. Veber // Proc. 2nd IEEE Conference on Industrial Informatics (INDIN'04). – Berlin, Germany, 2004. – P. 612–617.

133. Fowler, M. Refactoring: Improving the Design of Existing Code / M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. – Addison-Wesley, 1999. – 420 c.

134. Function Block Development Kit. – URL: http://www.holobloc.com/doc/fbdk/index.htm, 2008.

135. Gašević, D. Model Driven Architecture and Ontology Development / D. Gašević, D. Djurić, V. Devedžić. – Berlin: Springer, 2005. - 312 p.

136. Gawanmeh, A. An Executable Operational Semantics for SystemC using Abstract State Machines / A. Gawanmeh, A. Habibi, S. Tahar // Technical Report, Concordia University, Department of Electrical and Computer Engineering, 2004. – 24 c.

137. Gerber, C. Formal modelling of IEC 61499 function blocks with integer-valued data types / C. Gerber, I. Ivanova-Vasileva, H.-M. Hanisch // Control and Cybernetics. – 2010. – Vol. 39. – P. 197–231.

138. Glässer, U. Abstract State Machine Semantics of SDL / U. Glässer, R. Karges // Journal of Universal Computer Science. – 1997. – Vol. 3, № 12. – P. 1382–1414.

139. Glässer, U. Combining abstract state machines with predicate/transition nets / U. Glässer // Lecture Notes in Computer Science. – 1997. – Vol. 1333. – P. 108–122.

140. Glausch, A. Distributed Abstract State Machines and Their Expressive Power / A. Glausch, W. Reisig // Informatik-Berichte 196, Berlin : Humboldt-Universität, 2006. – 33 p.

141. Goh, K. M. Iterative Knowledge Based Code Generator for IEC 61499 Function Block / K. M. Goh, W. Ding, B. Tjahjono // 2009 IEEE Region 10 Conference (TENCON 2009). – Singapore, 2009. – P. 1–6.

142. Plotkin, G. D. A Structural Approach to Operational Semantics / G. D. Plotkin // DAIMI FN-19, Computer Science Department, Aarhus University. – 1981. – 133 p.

143. Grau, B. C. OWL2: The Next Step for OWL / B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, U. Sattler // Journal of Web Semantics : Science, Services and Agents on the World Wide Web. – 2008. – № 6(4). – P. 309–322.

144. Grunske, L. Graph Transformation for Practical Model Driven Software Engineering / L. Grunske, L. Geiger, A. Zuendorf, N. V. Eetvelde, P.V. Gorp, D. Varro // Model-driven Software Development. – Berlin; Heidelberg : Springer Verlag, 2005. – P. 91–118.

145. Guizzardi, G. Ontology-Based Evaluation and Design of Domain-Specific Visual Modeling Languages / G. Guizzardi, L. F. Pires, M. v. Sinderen // 14th International Conference on Information Systems Development. – Karlstad, Sweden : Springer, 2005.

146. Gunter, C. A. The Semantics of Programming Languages. Structures and Techniques / C. A. Gunter. – MIT Press, 1992. – 441 c.

147. Gurevich, Y. Evolving Algebras 1993: Lipari Guide / Y. Gurevich // Specification and Validation Methods. – Oxford : University Press, 1995. – P. 9–36.

148. Hagge, N. Modeling and clarifying the execution of IEC 61499 function blocks using XNet / N. Hagge, B. Wagner // Proc. IEEE 5th Int. Conf. Ind. Informat. (INDIN 2007). – Vienna, Austria, 2007. – P. 1177–1182.

149. Handbook of Graph Grammars and Computing by Graph Transformations / G. Rozenberg (ed.) // World Scientific. – 1997. – Vol. 1. – 553 p.

150. Hanisch, H.-M. Netz-Condition/Event Systeme / H.-M. Hanisch, M. Rausch // 4. Fachtagung Entwurf komplexer Automatisierungssysteme (EKA'95). – Braunschweig, 1995. – S. 55–71.

151. Hanisch, H.-M. One Decade of IEC 61499 Modeling and Verification – Results and Open Issues / H.-M. Hanisch, M. Hirsch, D. Missal, S. Preuße, C. Gerber // 13th IFAC Symposium on Information Con-

trol Problems in Manufacturing. – Moscow, 2009. – Vol. 13, Part 1, P. 211–216.

152. Happel, H. J. Applications of ontologies in software engineering / H. J. Happel, S. Seedorf // Proc. 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006). – Athens, USA, 2006.

153. Heckel, R. Confluence of Typed Attributed Graph Transformation Systems / R. Heckel, J. Malte Küster, G. Taentzer // Proceedings of the First International Conference on Graph Transformation (ICGT 2002). – 2002. – P. 161–176.

154. Hennessy, M. The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics / M. Hennessy. – Wiley, 1990. – 157 p.

155. Hermit OWL Reasoner. – URL: http://www.hermit-reasoner.com/

156. Heverhagen, T. A profile for integrating function blocks into the Unified Modeling Language / T. Heverhagen, R. Tracht, R. Hirschfeld // Int. Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS'03). – San Francisco, California, 2003.

157. Hirsch, M. Systemspezifikation mit SysML für eine Fertigungstechnische Laboranlage / M. Hirsch, H.-M. Hanisch // Fachtagung zum Entwurf komplexer Automatisierungssysteme (EKA 08). – Magdeburg, Germany, 2008. – P. 23–34.

158. Hoare, T. Grand challenges for computing research / T. Hoare, R. Milner // The Computer Journal. – 2005. – Vol. 48. – P. 49–52.

159. Horrocks, I. The Even More Irresistible SROIQ / I. Horrocks, O. Kutz, U. Sattler // Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). – AAAI Press, 2006. – P. 57–67.

160. ICSTriplex ISaGRAF v. 6. – URL: http://www.isagraf.com, 2012.

161. IMS – Intelligent Manufacturing Systems. Global Research and Business Innovation Program. – URL: http://www.ims.org/

162. International Standard IEC 61131-3 (edition 2.0): Programmable Controllers / International Electrotechnical Commission. – Geneva, 2003. – 230 p.

163. International Standard IEC 61499. Function blocks for industrial-process measurement and control systems (edition 2.0). Part 1:

Architecture / International Electrotechnical Commission. – Geneva, 2005. – 245 p.

164. International Standard IEC 61499. Function blocks for industrial-process measurement and control systems (edition 2.0). Part 2: Software tool requirements: ed2.0 / International Electrotechnical Commission. – Geneva, 2012. – 97 p.

165. IPSI-XQ – The XQuery Demonstrator. – URL: http://ipsixq.sourceforge.net/

166. Ivanova-Vasileva, I. Transformation of IEC 61499 control systems to formal models / I. Ivanova-Vasileva, C. Gerber, H.-M. Hanisch // Int. Conf. «Automatics and Informatics (CAI`07)». – Sofia. – 2007. – P. 5–10.

167. Jammes, F. Service-Oriented Paradigms in Industrial Automation / F. Jammes, H. Smit // IEEE Transaction on Industrial Informatics. – 2005. – № 1(1). – P. 62–70.

168. Khalgui, M. A behavior model for IEC 61499 function blocks / M. Khalgui, X. Rebeuf, F. Simonot-Lion // Third Workshop on Modelling of Objects, Components and Agents (MOCA). – Aarhus, Denmark, 2004. – P. 71–88.

169. Khalgui, M. Reconfiguration Protocol for Multi-Agent Control Software Architectures / M. Khalgui, H.-M. Hanisch // IEEE Transactions on Systems Man and Cybernetics. Part C-Applications and Reviews. – 2011. – Vol. 41. – P. 70–80.

170. Klein, S. A Petri Net based Approach to the Development of correct Logic Controllers / S. Klein, G. Frey, L. Litz // Second International Workshop on Integration of Specification Techniques for Applications in Engineering (INT'02). – Grenoble, France, 2002. – P. 116–129.

171. Koehler, H. J. Integrating UML diagrams for production control systems / H. J. Koehler, U. Nickel, J. Niere, A. Zuendorf // 22nd Int. Conf. on Software Engineering (ICSE 2000). – Limerick, Ireland, 2000. – P. 241–251.

172. Lastra, J. L. M. An IEC 61499 Application generator for Scan-Based Industrial Controllers / J. L. M. Lastra, L. Godinho, A. Lobov, R. Tuokko // Proc. 3rd IEEE Conference on Industrial Informatics. – Perth, Australia, 2005. – P. 80–85.

173. Ledeczi, Á. Composing Domain-Specific Design Environments / Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, G. Karsai // Computer. –2001. – Vol. 34, № 11. – P. 44–51.

174. Lewis, R.W. Modelling control systems using IEC 61499: Applying function blocks to distributed systems / R.W. Lewis. – London : IET, 2001. – 192 p.

175. Lueder, A. Formal models for the verification of IEC 61499 function block based control applications / A. Lueder, C. Schwab, M. Tangermann, J. Peschke // IEEE Int. Conf. Emerging technologies and factory automation (ETFA'2005). – Catania. – 2005. – P. 105–112.

176. McMillan, K. L. Symbolic Model Checking / K. L. McMillan // Kluwer Academic Publishers, 1993. – 194 p.

177. McMillan K. L. The SMV language / K. L. McMillan. – Cadence Berkeley Labs. – Berkeley, USA. – 111 p.

178. MEDEIA – Model-Driven Embedded System Design Environment for the Industrial Automation Sector. – URL: http://www.medeia.eu

179. Mens, T. On the Use of Graph Transformations for Model Refactoring / T.Mens // Lecture Notes in Computer Science. Generative and Transformational Techniques in Software Engineering. – 2006. – Vol. 4143. – P. 219–257.

180. Mir, A. A. Modeling and verification of embedded systems using Cadence SMV / A. A. Mir, S. Balakrishnan, S. Tahar // Canadian Conference on Electrical and Computer Engineering. – Halifax, 2000. – Vol. 1. – P. 179–183.

181. Missal, D. Modular Plant Modelling for Distributed Control / D. Missal, H.-M. Hanisch // Proceedings of 2007 IEEE International Conference on Systems, Man and Cybernetics. – Montréal, Canada, 2007. – P. 3475–3480.

182. Model-Driven Software Development / B. Sami, M. Book, V. Gruhn (Eds.). – London : Springer, 2005. – 464 p.

183. Mosses, P. D. Formal Semantics of Programming Languages An Overview / P. D. Mosses // Electronic Notes in Theoretical Computer Science. – 2006. – Vol. 148, Issue 1. – P. 41–73.

184. Motik, B. Query answering for OWL DL with rules / B. Motik, U. Sattler, R. Studer // Journal of Web Semantics. – 2005. – № 3(1). – P. 41–60.

185. Mueller, W. The semantics of behavioral VHDL'93 descriptions / W. Mueller, E. Boerger, U. Glaesser // Proceedings of the conference on European design automation (EURO-DAC '94). – Grenoble, 1994. – P. 500–505.

186. NuSMV – New Symbolic Model Checker. – URL: http:// nusmv.irst.itc.it

187. nxtStudio – Engineering-Software für alle Aufgaben. – URL: http://www.nxtcontrol.com/produkte/nxtstudio.html, 2012.

188. OOONEIDA: An Open Object-Oriented Knowledge Economy for Intelligent Industrial Automation. – URL: http://www.oooneida.net

189. OOONEIDA. IEC 61499 Compliance Profile: Execution Models of IEC 61499 Function Block Applications, draft in progress. – URL: http://www.oooneida.org/standards_development_Compliance_Profile.html, 2009.

190. Object Management Group (OMG). – URL: http://www.omg.org

191. OMG Systems Modeling Language. – URL: http://www.omg-sysml.org/

192. OMG's MetaObject Facility. – URL: http://www.omg.org/mof/

193. Ontologies for Software Engineering and Software Technology / Calero, Coral; Ruiz, Francisco; Piattini, Mario (Eds.). – London : Springer, 2006. – 339 p.

194. Orozco, O. J. L. Adding Function Blocks of IEC 61499 Semantic Description to Automation Objects / O. J. L. Orozco, J. L. M. Lastra // IEEE Conference on Emerging Technologies and Factory Automation (ETFA '06). – Prague, 2006. – P. 537–544.

195. O'Sullivan, D. VHDL architecture for IEC 61499 function blocks / D. O'Sullivan, D. Heffernan // Computers & Digital Techniques. – 2010. – Vol. 4. – P. 515–524.

196. Panjaitan, S. Combination of UML modeling and the IEC 61499 function block concept for the development of distributed automation systems / S. Panjaitan, G. Frey // IEEE Conference on Emerging Technologies and Factory Automation. – 2006. – P. 449–456.

197. Pang, C. Towards Formal Verification of IEC61499: modelling of Data and Algorithms in NCES / C. Pang, V. Vyatkin // IEEE Int. Conf. on Industrial Informatics (INDIN'2007). – Vienna, 2007. – P. 879–884.

198. Pellet – an open-source Java OWL DL reasoner. – URL: http://pellet.owldl.com/.

199. Podgurski, A. A Formal Model of Program Dependence and its Implication for Software Testing, Debugging and Maintenance / A. Podgurski, L. A. Clarke // IEEE Transactions on Software Engineering. – 1990. – № 16 (9). – P. 965–979.

200. Proceedings of 2nd International Workshop on Ontology-Driven Software Engineering (ODiSE'10). – Nevada, New York : ACM, 2010.

201. Protégé. –URL: http://protege.stanford.edu.

202. Ramadge, P. J. Supervisory control of a class of discrete-event processes / P. J. Ramadge, W. M. Wonham // SIAM Journal Control and Optimization. – 1987. – Vol. 25, № 1. – P. 206–230.

203. Rausch, M. Netz-Condition/Event-Systeme / M. Rausch, H.-M. Hanisch // 4. Fachtagung Entwurf komplexer Automatisierungssysteme. – Braunschweig, Germany, 1995. – P. 55–71.

204. Rausch, M. Net condition/event systems with multiple condition outputs / M. Rausch, H.-M. Hanisch // IEEE Int. Conf. on Emerging Technologies and Factory Automation, Paris, 1995. – Vol. 1. – P. 592–600.

205. Ryssel, U. Generation of Function Block Based Designs using Semantic Web Technologies / U. Ryssel, H. Dibowski, K. Kabitzsch // IEEE International Conference on Emerging Technologies and Factory Automation. – 2009. – P. 1–8.

206. Schuer, A. Graph Grammar Engineering with PROGRES / A. Schuer, A. J. Winter, A. Zuendorf // Proc. 5th European Software Engineering Conf. (ESEC'95), Lecture Notes in Computer Science. – Berlin : Springer Verlag, 1995. – Vol. 989. – P. 219–234.

207. Selic, B. Using UML for complex real-time systems / B. Selic, J. Rumbaugh. – URL: http://www.objecttime.com/technical/umlrt.html

208. Sendall, S. Model transformation: The heart and soul of model-driven software development / S. Sendall, W. Kozaczynski // IEEE Software. Special Issue on Model-Driven Software Development. – 2003. – № 20(5). – P. 42–45.

209. SESA: Signal-Net System Analyzer. – URL: http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/tools/.

210. Shasha, D. Algorithmics and Applications of Tree and Graph Searching / D. Shasha, J. T. L. Wang, R. Giugno // Proc. 21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems. – New York, USA, ACM Press, 2002. – P. 39–52.

211. SHE/POOSL Website. – URL: http://www.es.ele.tue.nl/poosl

212. Sinha, R. Observer based verification of IEC 61499 function blocks / R. Sinha, P. S. Roop // 9th IEEE International Conference on Industrial Informatics (INDIN). – 2011. – P. 609–614.

213. Sowa, J. F. Principles of Semantic Networks: Explorations in the Representation of Knowledge / J. F. Sowa, A. Borgida // Elsevier Science & Technology Books. – 1991. – 500 p.

214. Stafford, J. A. Architecture-Level Dependence Analysis for Software Systems / J. A. Stafford, A. L. Wolf // Int. Journal of Software

Engineering and Knowledge Engineering. – 2001. – Vol. 11, № 4. – P. 431–452.

215. Stanica, P. Using Timed Automata for the Verification of IEC 61499 Applications / P. Stanica, H. Gueguen // IFAC Workshop on Discrete Event Systems (WODES'04). – Reims, France, 2004. – P. 22–24.

216. Starke, P. H. Analysing Signal-Net Systems / P. H. Starke, S. Roch // Informatik–Bericht 162. – Humboldt-Universitaet zu Berlin, Institut fuer Informatik, 2002. – 136 p.

217. Streitferdt, D. Model Driven Development Challenges in the Automation Domain / D. Streitferdt, G. Wendt, P. Nenninger, A. Nyen, L. Horst // 32nd Annual IEEE International Computer Software and Applications Conference. – 2008. – P. 1372–1375.

218. Sünder, C. Execution Models for the IEC 61499 elements Composite Function Block and Subapplication / C. Sünder, A. Zoitl, J.H. Christensen, M. Colla, T. Strasser // 5th IEEE Int. Conference on Industrial Informatics (INDIN'07). – Vienna, Austria, 2007. – P. 1169–1175.

219. Sünder, C. Usability and Interoperability of IEC 61499 based distributed automation systems / C. Suender, A. Zoitl A., J.H. Christensen, V. Vyatkin, R. Brennan, A. Valentini, L. Ferrarini, K. Thramboulidis, T. Strasser, J. L. Martinez-Lastra, F. Auinger // Proc. 4th IEEE Conference on Industrial Informatics (INDIN'06). – Singapore, 2006. – P. 31–37.

220. SWI Prolog. – URL: http://www.swi-prolog.org/

221. SWRL: A Semantic Web Rule Language. W3C Member Submission, 2004. – URL: http://www.w3.org/Submission/SWRL/

222. Taenzer, G. AGG: A Tool Environment for Algebraic Graph Transformation / G. Taenzer // Lecture Notes in Computer Science. – 2000. – Vol. 1779. – P. 481–490.

223. Tata, P. Proposing a novel IEC61499 Runtime Framework implementing the Cyclic Execution Semantics / P. Tata, V. Vyatkin // 7th IEEE International Conference on Industrial Informatics. – 2009. – P. 416–421.

224. The Description Logic Handbook: Theory, Implementation, and Applications / F. Baader [et al.]. – London : Cambridge University Press, 2003. – 574 p.

225. Thramboulidis, K. Development of Distributed Industrial Control Applications: The CORFU Framework / K. Thramboulidis // 4th IEEE International Workshop on Factory Communication Systems. – Sweden, 2002. – P. 39–46.

226. Thramboulidis, K. IEC 61499 in Factory Automation / K. Thramboulidis // Advances in Computer, Information, and Systems Sciences, and Engineering. – 2006. – P. 115–124.

227. Thramboulidis, K. Model-Integrated Mechatronics – Toward a New Paradigm in the Development of Manufacturing Systems / K. Thramboulidis // IEEE Transactions on Industrial Informatics. – 2005. – Vol. 1, Issue 1. – P. 54–61.

228. Thramboulidis, K. Using UML in Control and Automation: A Model Driven Approach / K. Thramboulidis // 2nd IEEE International Conference on Industrial Informatics (INDIN'04). – Berlin, Germany, 2004. – P. 587–593.

229. TNCES Workbench. – URL: http://sourceforge.net/projects/tnces-workbench/

230. Tranoris, C. Integrating UML and the function block concept for the development of distributed control applications / C. Tranoris, K. Thramboulidis // 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003). – Lisbon, Portugal, 2003. – Vol. 2. – P. 87–94.

231. Uppaal – an integrated tool environment for modeling, validation and verification of real-time systems. – URL: http://www.uppaal.org/

232. Varea, V. Dual transitions Petri net based modelling technique for embedded system specification / V. Varea, B. Al-Hashimi // Conference on Design, automation and test in Europe. – Munich, Germany, 2001. – P. 566–571.

233. Vidal, C. A High-level Petri Net Ontology Compatible with PNML / C. Vidal, M. Lama, A. Bugarin // 2006 Petri Net Markup Language Forum, Turku, Finland, 26 June, 2006. – P. 2–23.

234. ViVe – VisualVerifier Tool Framework. – URL: http://www.fb61499.com/license.html

235. Vyatkin, V. A modeling approach for verification of IEC1499 function blocks using Net Condition/Event Systems / V. Vyatkin, H.-M. Hanisch // IEEE conference on Emerging Technologies in Factory Automation (ETFA'99). – Barcelona, 1999. – P. 261–270.

236. Vyatkin, V. Alternatives for Execution Semantics of IEC61499 / V. Vyatkin, V. Dubinin, C. Veber, L. Ferrarini // 5th IEEE International Conference on Industrial Informatics (INDIN'2007). – Vienna, Austria, 2007. – P. 1105–1110.

237. Vyatkin, V. Closed-Loop Modeling in Future Automation System Engineering and Validation / V. Vyatkin, H.-M. Hanisch, P. Cheng,

Y. Chia-Han // IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews. – 2009. – Vol. 39. – P. 17–28.

238. Vyatkin, V. Execution Model of IEC61499 Function Blocks based on Sequential Hypothesis / V. Vyatkin, V. Dubinin. – URL: http://www.ece.auckland.ac.nz/~vyatkin/o3fb/VyatkinDubinin_SequentialSemantic.pdf

239. Vyatkin, V. Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems / V. Vyatkin // Proc. 5th IEEE Int. Conf. Industrial Informatics (INDIN), 2006. – P. 874–879.

240. Vyatkin, V. Formalisms For Verification Of Discrete Control Applications On Example Of IEC 61499 Function Blocks / V. Vyatkin, H.-M. Hanisch, P. Starke, S. Roch // Fachtagung Verteilte Automatisierung, Magdeburg, Germany, 2000. – P. 72–79.

241. Vyatkin, V. IEC 61499 as Enabler of Distributed and Intelligent Automation: State of the Art Review / V. Vyatkin // IEEE Transactions on Industrial Informatics. – 2011. – Vol. 7, Issue 4. – P. 768–781.

242. Vyatkin, V. IEC 61499 function blocks for embedded and distributed control systems design / V. Vyatkin. – ISA, 2007. – 279 p.

243. Vyatkin, V. On Comparison of the ISaGRAF implementation of IEC 61499 with FBDK and others implementations / V. Vyatkin, J. Chouinard // 4th IEEE Int. Conference on Industrial Informatics (INDIN'06). – 2006. – P. 1169–1175.

244. Vyatkin, V. OOONEIDA: an open, object-oriented knowledge economy for intelligent industrial automation / V. Vyatkin, J. H. Christensen, J. L. M. Lastra // IEEE Transactions on Industrial Informatics. – 2005. – Vol. 1. – P. 4–17.

245. Vyatkin, V. Rapid engineering and re-configuration of automation objects using formal verification / V. Vyatkin, H.-M. Hanisch, S. Karras, T. Pfeiffer, V. Dubinin // International Journal of Manufacturing Research. – 2006. – Vol. 1, № 4. – P. 382–404.

246. Vyatkin, V. Refactoring of Execution Control Charts in Basic Function Blocks of the IEC 61499 Standard / V. Vyatkin, V. Dubinin // IEEE Transactions on Industrial Informatics. – 2010. – Vol. 6, Issue 2. – P. 155–165.

247. Vyatkin, V. Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499 / V. Vyatkin, V. Dubinin // Proc. 5th IEEE Conference on Industrial Informatics (INDIN'07). – Vienna, Austria, 2007. – P. 1137–1142.

248. Vyatkin, V. The IEC 61499 Standard and Its Semantics / V. Vyatkin // IEEE Industrial Electronics Magazine. – 2009. – Vol. 3. – P. 40–48.

249. Vyatkin, V. Verification of distributed control systems in intelligent manufacturing / V. Vyatkin, H.-M. Hanisch // Journal of Intelligent Manufacturing. – 2003. – № 14 (1). – P. 123–136.

250. W3C Консорциум. Язык Web-онтологий OWL. – URL: http://www.w3.org/2004/OWL/

251. Walter, T. OntoDSL: An Ontology-Based Framework for Domain-Specific Languages / T. Walter, F.S. Parreiras, S. Staab // Proc. 12th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2009). – Denver, USA, 2009. – P. 408–422.

252. Wimmel, G. A BDD-based Model Checker for the PEP Tool / G. Wimmel. – Major Individual Project Report, University of Newcastle. – Newcastle, 1997. – 107 p.

253. Winskel, G. The Formal Semantics of Programming Languages: An Introduction / G. Winskel. – MIT Press, 1993. – 360 p.

254. Winter, K. Model Checking for Abstract State Machines / K. Winter // Journal of Universal Computer Science. – 1997. – Vol. 3. – № 5. – P.689–701.

255. Wongthongtham, P. Development of a Software Engineering Ontology for Multisite Software Development / P. Wongthongtham, E. Chang, T. Dillon, I. Sommerville // IEEE Transactions on Knowledge and Data Engineering. – 2009. – Vol. 21, № 8. – P. 1205–1217.

256. Yoo, J. A Verification Framework for FBD Based Software in Nuclear Power Plants / J. Yoo, S. Cha, E. Jee // 15th Asia-Pacific Software Engineering Conference, (APSEC '08). – Beijing, 2008. – P. 385–392.

257. Yoong, L. H. A Synchronous Approach for IEC 61499 Function Block Implementation / L. H. Yoong, P. Roop, V. Vyatkin, Z. Salcic // IEEE Transactions on Computers. – 2009. – Vol. 58 (12). – P. 1599–1614.

258. Yoong, L. H. Synthesizing Globally Asynchronous Locally Synchronous Systems With IEC 61499 / L. H. Yoong, G. D. Shaw, P. S. Roop, Z. Salcic // IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews. – 2012. – Volume: PP, Issue: 99. – P. 1–13.

259. Yoong, L. H. Verifying IEC 61499 Function Blocks Using Esterel / L. H. Yoong, P. Roop // IEEE Embedded systems letters. – 2010. – Vol. 2, № 1. – P. 1–4.

260. Zagar, K. The control system modeling language / K. Zagar, M. Plesko, M. Sekoranja, G. Tkacik, A. Vodovnik // Proc. 8th Int. Conf. On Accelerator&Large Experimental Physics Control Systems (ICALEPCS 2001). – San Jose, California, 2001. – P. 472–474.

261. Zhang, W. Comparison between Function Block-Oriented and Object-Oriented Design in Control Applications / W. Zhang, C. Diedrich, W. Halang // 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP 2003). – Lagow, Poland, 2003.

262. Zoitl, A. Executing real-time constrained control applications modelled in IEC 61499 with respect to dynamic reconfiguration / A. Zoitl, G. Grabmair, F. Auinger, C. Suender // Proc. 3rd IEEE Conference on Industrial Informatics (INDIN'05). – Perth, Australia, 2005. – P. 62–67.

263. Zoitl, A. IEC 61499 Architecture for Distributed Automation: The "Glass Half Full" View" / A. Zoitl, V. Vyatkin // IEEE Industrial Electronics Magazine. – 2009. – Vol. 3. – P. 7–22.

264. Zoitl A. Real-Time Execution for IEC 61499 / A. Zoitl. – ISA, 2009. – 276 p.

265. Zoitl, A. The past, present, and future of IEC 61499 / A. Zoitl, T. Strasser, K. Hall, R. Staron, C. Sunder, B. Favre-Bulle // 3rd Int. Conf. on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS 2007). – Regensburg, Germany, 2007. – P. 1–14.

V. Dubinin, V. Vyatkin

Function blocks of IEC 61499 standard: validation and transformation in design of distributed automation systems

Abstract

This book lays the foundation of the theory and technology of designing distributed component-based control for a new generation of industrial automation systems based on the IEC 61499 international standard.

Operational semantics of IEC 61499 function blocks for different models of execution is considered along with formal verification method and a method of semantic analysis of IEC 61499 projects using *Web*-ontologies. A unified approach to the design of control systems based on graph transformation is proposed. This approach solves the problem of formal models' synthesis, refactoring, and portability of industrial automation software.

The book can be interesting to experts in the field of computer technology, automation and robotics.

Summary

Modern control systems in industrial automation (typically constructed based on the international standard IEC 61131-3) have centralized architecture which implies a number of shortcomings and limitations, such as reliability and performance bottlenecks, the complexity of configuration and support, the complexity of modifications (build, remove and modify components) for the construction of reconfigurable systems, etc. At the same time, more stringent conditions in which the market puts producers of industrial goods (including the shift from mass production to mass customization, the need for quick reconfiguration of the equipment on the production of new products, the complexity of the products, increasing demands for them, etc.), leading them to transition to a new technological basis, which allows to reduce design and redesign control systems and have the ability of quick reconfiguration.

A new international standard IEC 61499, adopted in 2005, aims at addressing those challenges by proposing a reference architecture for distributed control industrial processes. In essence, the standard introduces a class of control systems of the new generation that are intelligent reconfigurable distributed component-based system. The IEC 61499 standard supports modular design paradigm based on function blocks (FB). This

paradigm has incorporated the features of the component, object-oriented and automata approach to the design and programming of complex automation and control systems.

There many research works published worldwide on the topics related to IEC 61499, however, many problems related to the standard are still unresolved. Among them are the following: 1) the development of methods and tools for the design and re-design of distributed control systems based on IEC 61499; 2) the development of suitable execution models and software portability methods, and 3) migration of projects based on the "old" *PLC* standard IEC 61131-3 to the new platform of IEC 61499.

The aim of this work is the development of effective methods of reliable control systems design for the new generation of industrial processes.

This book is structured as follows. The *first section* provides analysis of control systems design methods based on the IEC 61499 standard, including the formal description techniques and design methodologies. The section provides an overview of trends in the development of modern industrial automation systems, the challenges in this area, the requirements for control systems. An overview of the IEC 61499 standard is also provided including discussion of its advantages, features and problems.

The *second section* is devoted to the development and use of *UML-FB* - a visual language for modeling of industrial process control on basis of the standard IEC 61499. This language is based on a "lightweight" extension of *UML* meta-model using the mechanism of stereotypes that reduces semantic gap between the structured and object-oriented design approaches. An example of *UML-FB* specification of *FESTO* laboratory-scale production system confirmed the suitability and usability of *UML-FB*.

The *third section* defines operational semantics of FB. As a prerequisite provision, the deployment of a system configuration to the level of instances and data buffers is considered. The main part of the section suggests two FB semantics: the first is based on the abstract state machines (*ASM*), and the second - on the state-transition systems (*STS*).

In the *fourth section* the model checking technique is adapted for formal verification of FB systems. The techniques are proposed for creating formal models of FB systems using the language of *SMV* verifier. Two approaches are proposed, for both operational semantics considered in the previous section.

The *fifth section* is devoted to developing graph transformational approach to the synthesis of formal models of function block systems. The arithmetic Net Condition-Event Systems (*aNCES*) is used as formal model. The section presents basics of the theoretical framework of graph transformations, and proposes a generalized graph for the models flow used in the synthesis of FB systems net models.

The *sixth section* discusses refactoring of Execution Control Chart (*ECC*) state machines encapsulated to basic function blocks to get rid from potential deadlocks. Under this approach, a re-factoring software tool is developed based on *ECC* graph meta-model and the transformation rules of graphs rewriting.

The *seventh section* defines the ontological approach to semantic analysis of control systems based on the IEC 61499 standard. An ontology of FB built using descriptive logic and Horn clauses is described. As part of the approach the following are developed: a) the specific semantic constraints to identify many semantic errors in the description of control systems, and b) the method of detecting dangerous cycles for hierarchical structures in FB systems. The implementation of the FB ontology based on the languages *OWL DL* and *SWRL* in the *Protégé* tool is discussed. The method of dangerous cycles detection is demonstrated on an example. Semantic analysis of IEC 61499 projects with *Web*-ontologies is proposed.

The *eighth section* sets out the approach to the problem of portability for automation software that is built in accordance with IEC 61499. The approach relies on model-driven implementation patterns of FB systems. We consider the transformation of the basic FBs, including the interface and *ECC* charts, and of composite FBs, which is largely determined by the selected execution model. Two implementation patterns focused on cyclic and synchronous execution models are described and illustrated on examples.

Научное издание

Дубинин Виктор Николаевич,
Вяткин Валерий Владимирович

Модели функциональных блоков IEC 61499, их проверка и трансформации в проектировании распределенных систем управления

Редактор *В. В. Чувашова*
Корректор *Ж. А. Лубенцова*
Компьютерная верстка *М. Б. Жучковой*

Подписано в печать 19.12.12.
Формат 60×84 1/16. Усл. печ. л. 20,23.
Тираж 200. Заказ № 1074.

Издательство ПГУ.
440026, Пенза, Красная, 40.
Тел./факс: (8412) 56-47-33; e-mail: iic@pnzgu.ru

Дубинин Виктор Николаевич – к.т.н., доцент кафедры вычислительной техники Пензенского государственного университета. В 1981 г. с отличием окончил Пензенский политехнический институт, в 1989 г. защитил кандидатскую диссертацию по специальности «Вычислительные машины, комплексы, системы и сети» в Рязанском радиотехническом институте. В 2003, 2006, 2010 гг. работал как приглашенный ученый в рамках грантов DAAD в университете Мартина Лютера (Германия), а в 2011 г. – в университете Окленда (Новая Зеландия). Область научных интересов – формальные методы для спецификации, верификации, синтеза и реализации распределенных и дискретных событийных систем.

Вяткин Валерий Владимирович – д.т.н., профессор, заведующий кафедрой "Ответственные вычисления и коммуникации" Технического универстета Лулео (Швеция) и визит-профессор Кембриджского университета (Великобритания). Ранее занимал должность профессора в университете Окленда (Новая Зеландия), где возглавлял лабораторию инфомехатроники и промышленной автоматики. В 1988 г. с отличием окончил Таганрогский радиотехнический институт, в 1992 и 1998 гг. там же защитил кандидатскую и докторскую диссертации. С 1994 по 2002 г. работал в Нагойском технологическом институте (Япония) и в университете Мартина Лютера (Германия). Область научных интересов – программное обеспечение систем промышленного управления и информатики, включая распределённые программные архитектуры, инженерию программного обеспечения, мультиагентные и реконфигурируемые системы, приложения искусственного интеллекта в промышленной автоматике. Является членом стандартизационного комитета Международной электротехнической комиссии (МЭК) по стандартам IEC 61131 и IEC 61499, был одним из инициаторов и создателей международной организации O3NEIDA.

ISBN 978-5-94170-521-4

ПензГУ