САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

На правах рукописи

Соколов
Владимир Владимирович

СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ MSC И SDL МОДЕЛЕЙ ПРИ РАЗРАБОТКЕ СОБЫТИЙНО-ОРИЕНТИРОВАННЫХ СИСТЕМ

05.13.11 — математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени кандидата физико-математических наук

Научный руководитель —
доктор физико-математических наук, профессор А.Н.Терехов

Санкт-Петербург
2007

Оглавление

Введение

В последнее десятилетие в развитии программного обеспечения произошел качественный скачок. Электронный мир теперь распределен, параллелен и взаимосвязан. Распределен — поскольку информация размещена во многих местах по всему миру. Параллелен — поскольку деятельность происходит во многих местах одновременно. Одного потока управления теперь недостаточно ни для принятия делового решения, ни для выполнения программ. Взаимосвязан — поскольку действие в одном месте может значительно повлиять на окружающий мир. Возрастает число компаний, доверяющих критически значимые операции системам распределенных объектов предприятия. Эти объектно-ориентированные системы содержат распределенные серверы и базы данных, объединенные для поддержки высокопараллельных бизнес-операций. Все большее число отраслей для осуществления своей деятельности вынуждены взаимодействовать в режиме реального времени. Банки, авиалинии, телефонные компании не могут работать без интенсивного обмена информацией между всеми действующими объектами. Наконец, устройства реального времени встречаются теперь повсюду — в машинах, приборах, бытовой электронике, зданиях. Большинство современных систем являются системами реального времени. Проблемы их работы перестали быть узкой специализированной областью и касаются теперь каждого из нас. Новый электронный мир несет с собой большие возможности, но они даются ценой высоких затрат на разработку. Основой сосуществования параллельных распределенных систем являются протоколы их взаимодействия. Простые компьютерные системы, языки и модели прошлого не могут соответствовать современным требованиям. С возрастанием сложности протоколов взаимодействия потребовались специальные

средства, нацеленные на описание, разработку и отладку соответствующих протоколов.

Разумеется, системы, в которых использовались сложные протоколы взаимодействия, существовали достаточно давно, и проблемы, которые сейчас встали перед всей компьютерной областью, там проявились гораздо раньше. Мы говорим о телекоммуникационных системах. Соответствующие проблемы там были успешно разрешены, в результате чего были созданы стандарты SDL (Specification and Description Language) [48] и MSC (Message Sequence Chart) [49]. Данные стандарты были разработаны Международным Консультационным Комитетом по Телеграфии и Телефонии, ныне ITU-T [27]. На русском языке стандарты были описаны в работах [7, 12, 4].

Конечно, данные языки не сразу достигли современного уровня, еще до получения статуса стандарта велась долгая предварительная работа, а после выпуска первой версии стандарты многократно перерабатывались и дополнялись. Оба стандарта являются хорошим балансом между аналитическими и выразительными возможностями. Это стало возможно только потому, что каждый из стандартов имеет как графическую, так и программную нотацию. Существенно, что между данными спецификациями нет никакой разницы и, при необходимости, можно использовать как один, так и другой вариант. Это дает возможность использовать данные стандарты как средство проектирования для обсуждения графических нотаций с экспертами предметной области, так и использовать их для программирования и тестирования систем.

Данные языки являются стандартом де-факто в международной телефонии. Организация ITU-T использует их для спецификации других стандартов (протоколов) — UMTS, GSM, E-DSS-1, V5.2, SS7 и других. MSC и SDL широко используются в известных фирмах–разработчиках телекоммуникационного оборудования, таких как Alcatel, Ericson, Hewlett-Packard, Motorola, Nokia, Siemens. При этом следует упомянуть, что изначальная область применения данных стандартов шире, чем телекоммуникации, например, язык SDL был выбран языком спецификаций компании Боинг [4]. В России эти стандарты используются при разработке и сертификации те-

лекоммуникационного оборудования [3].

В свете того, что все большее количество приложений становится распределенными, идет процесс интеграции данных языков в такую популярную методологию разработки программного обеспечения, как UML [25]. Изначально данный подход был создан авторами трех наиболее распространенных методологий Гради Буч (BOOCH), Джим Рамбо (OMT — Object Modelling Technique) и Айвар Якобсон (OOSE — Object Oriented Software Engineering) под эгидой Rational Software Corporation [26]. Он должен был объединить все существенные и успешные разработки в данной области и стать стандартом языка объектного моделирования. Наряду с Rational в его создании участвовали представители множества компаний, таких как Microsoft, IBM, Hewlett-Packard, Oracle, DEC, Unisys и нескольких сотен более мелких и завершился созданием в январе 1997 года версии 1.0. На текущий момент UML используют в качестве стандартов такие гранды как Microsoft, Hewlett-Packard, Oracle, SybaseLogic Works. Практически все мировые производители CASE систем (Computer Aided Software Engineering) либо уже поддерживают UML в своих продуктах, либо заявили об этом в своих планах. При этом реализованы переходы из UML в множество языков программирования, таких как C++, Java, Delphi, VisualBasic, Ada.

Данная работа нацелена на языки SDL и MSC, но при этом полученные результаты также могут быть применены и для UML, что позволяет говорить о широкой области применимости предложенных алгоритмов в разработке систем.

Возвращаясь к классической области применимости данных стандартов для создания телекоммуникационного оборудования, можно привести ряд любопытных оценок. Соответствующее ПО относится к категории сверхсложного, к которому предъявляются очень жесткие требования [3]:

- время поиска неисправности не больше 15 минут;
- время восстановления оборудования не больше 30 минут;
- наработка на отказ — от 10000 часов;
- время простоя не больше 2 часов за 40 лет;

Большáя часть этой сложности приходится на описание распределенности и возможных отказов различных частей системы. Поэтому важными проблемами данной области являются:

- технология создания больших систем;
- проверка корректности;
- обработка ошибочных ситуаций взаимодействия.

Коллектив кафедры системного программирования и ГУП "Терком" многие годы занимаются созданием телефонных станций. У нас есть собственная технология разработки на базе стандартов SDL, MSC, UML. Она прошла долгую эволюцию, и в различное время была известна под именами RTST [18, 14] /RTST++ [6, 5] /REAL [19, 16, 10]. При этом практическое использование данной технологии показало ее применимость не только для разработки телефонных станций, но и для других систем реального времени и информационных систем, в которых очень важен событийный аспект описания системы.

Решения, предлагаемые в данной работе, получены автором путем расширения данной технологии. Результаты получены на базе практического ее использования, однако они применимы для любой другой технологии, содержащей в себе модели MSC и SDL, например для [29], либо допускающих их использование в качестве промежуточной модели, таких как в работах [45, 37, 61, 52].

Введение в модели MSC и SDL и необходимость их совместного использования

SDL (Specification and Description Language) диаграмма очень похожа на традиционные блок-схемы, но с несколькими важными расширениями: символ состояния, в котором процесс не занимает процессор, ожидая приема одного или нескольких сообщений[1]; символ приема сообщения и символ

[1]В данной работе считаем понятие сигнала (signal), традиционно используемого в SDL, эквивалентным понятию сообщения (message), используемого в MSC.

посылки сообщения, как это изображено на рисунке 1. В SDL диаграммах основное внимание уделяется логике использования сообщений; логика выполнения действий, не связанных с сообщениями, детализируется лишь по мере необходимости. Поэтому действие может быть атомарным, как показано на рисунке 1, так и достаточно сложным из десятков или сотен операторов.

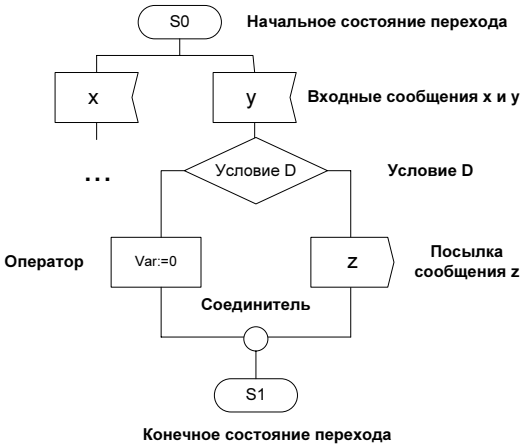

Рис. 1: Пример SDL диаграммы

MSC диаграммы (Message Sequence Chart) позволяют описывать сценарии поведения системы во времени. Время течет сверху вниз, вертикальные линии представляют объекты системы, а между ними рисуются стрелки, обозначающие сообщения. Элементарная MSC диаграмма изображена на рисунке 2. MSC диаграммы состоят из отдельных сценариев и структур, задающих последовательности выполнения этих сценариев. Они широко применяются для описания протоколов различными международными организациями, в том числе для стандартов, разработанных организацией ITU-T, однако редко когда приводятся исчерпывающие описания поведения системы, включая обработку аварийных ситуаций, техобслуживания, тарификации и т.д. [39, 40]. И MSC, и SDL диаграммы применяются для описания динамического поведения системы, их сравнительный анализ приведен в таблице 1. Используемые в данной работе элементы MSC и SDL диаграмм приведены в Приложении А.

В современных системах используются оба стандарта, поскольку они дают различную информацию о динамическом поведении объекта и до-

Таблица 1: Сравнительные характеристики SDL и MSC.

MSC	**SDL**
Описывает взаимодействие между различными частями системы; по отношению к объекту данный стандарт является внешним	Описывает внутренние алгоритмы каждого из объектов; по отношению к объекту стандарт является внутренним
Отображена логика взаимодействия нескольких объектов. Логика поведения каждого из них скрыта	Отображена логика поведения каждого из объектов. Логика их взаимодействия скрыта
Информация о выборе линии поведения дается словесно	Логика работы детализована вплоть до значений переменных
Создаются на этапе проектирования	Создаются при программировании
Используются при обсуждении задачи с экспертами предметной области	Преимущественно используются программистами

полняют друг друга. Подчеркнем, что они являются не просто различными представлениями одной и той же модели, а несут в себе разную информацию. Например, только из SDL диаграмм можно узнать о выборе одной либо другой ветви поведения. Соответствующая информация содержится в переменных, которые на MSC отсутствуют.

Для того, чтобы указать информацию, которая содержится только на MSC диаграммах, рассмотрим гипотетическую ситуацию с банком, когда для некоторых клиентов кассир сразу выдает деньги, а для других предва-

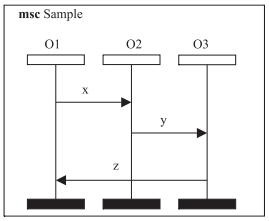

Рис. 2: Пример MSC диаграммы

рительно консультируется у директора банка. В общем случае, имея SDL модели всех участников, нельзя восстановить, участвовал ли директор в процедуре выдачи денег конкретному клиенту. Можно показать, что эта задача родственна задаче о самоприменимости и алгоритмически неразрешима. Соответствующая информация хранится на MSC диаграммах, которые содержат информацию не только о том, как объекты обмениваются сообщениями, но и кто участвует в сценариях взаимодействия.

Поэтому за какой-то информацией приходится обращаться к MSC диаграммам, а за какой-то — к SDL диаграммам. Из-за этого в современных технологических средствах присутствуют оба типа диаграмм. Соответственно, возникает необходимость обеспечения их согласованности.

Обсуждение области

Для технолога[2] было бы идеальным вариантом, если бы существовала некая объемлющая модель, на которую, как на трехмерный кубик, можно было бы взглянуть с одной стороны и увидеть SDL диаграммы, а с другой грани она бы давала представление в виде MSC диаграмм. На текущий момент времени подобная модель не существует, можно лишь выделить ряд подходов по согласованию MSC и SDL диаграмм в существующих CASE системах.

Обзор существующих подходов по совместному использованию MSC и SDL

Условно их можно разделить на следующие группы.

1. MSC и SDL независимы.

   (a) MSC и SDL представляют собой отдельные независимые типы диаграмм. В этом случае их несогласованность ведет к потенциальным ошибкам.

[2]Так мы называем разработчика, создающего продукт с помощью технологии.

(b) Переход от MSC диаграмм к SDL диаграммам осуществляется человеком, что повышает вероятность ошибки и очень часто приводит к неоптимальным решениям, лишнему и избыточному коду, особенно при длительной поддержке проекта.

2. Попытки создавать системы на основе MSC диаграмм с последующим синтезом SDL.

   (a) В случае, когда пытаются поддерживать весь цикл разработки на MSC, то это влечет за собой расширение стандарта MSC и перенос на MSC диаграммы кода, данных и других технических деталей [60]. После этого начинаются проблемы при увеличении количества сценариев и вытекающей нестыковке участков кода. Структура MSC малоприменима для задания на ней кода. MSC диаграммы больше отвечают на вопрос «Что может случиться?», чем на вопрос «Почему так случается?», вследствие чего они обладают избыточностью и мало приспособлены для использования на них “арифметических” данных.

   (b) “Односторонний синтез”. В этом случае один раз производится синтез SDL диаграмм по MSC диаграммам, а затем используются полученные структуры [62, 30, 60]. В таком случае внесение изменений вручную очень дорого стоит. Зачастую это приводит к полному отказу от MSC диаграмм, по которым был проведен синтез.

   (c) “Инкрементальные алгоритмы”. Стандарт MSC урезается до диаграмм, задающих только трассы[3]. Существуют алгоритмы [55], позволяющие добавить новую MSC спецификацию — трассу на SDL диаграммы. Данный подход работает лишь на специфическом классе задач из-за постоянного присутствия цикличности в реальных задачах. Для неурезанного стандарта MSC задача создания инкрементальных алгоритмов также рассматривалась, но была разрешена лишь для очень узкого набора из-

[3]Под трассой имеем ввиду цепочку сообщений.

менений [50].

Алгоритмы, используемые в пунктах 2.c и 2.b [62, 30], способны провести синтез не из любых MSC описаний. Алгоритм пункта 2.a способен на это всегда. Побочным эффектом этого алгоритма является то, что в процессе его работы данные могут быть преобразованы таким образом, что в получившейся SDL диаграмме будет тяжело узнать исходную MSC диаграмму. Сразу отметим, что невозможность осуществить синтез во многих разумных ситуациях является очень слабым местом, поэтому следует отдавать предпочтение вариантам алгоритма 2.a.

3. Алгоритмы сравнения используемых MSC и SDL диаграмм для проверки соответствия.

   (a) На основе MSC диаграмм строятся трассы, а затем полученные трассы "накладываются"[4] на SDL диаграммы [40, 39, 41]. Если трасса укладывается, то все хорошо. Если нет — это считается ошибочной ситуацией. Подобный способ недостаточно эффективен, поскольку добавление где-либо отладочного сообщения нарушает последовательность сообщений на трассе и считается, что SDL модель не соответствует MSC спецификациям [72]. Этим методом невозможно проверить случай, когда SDL модель является реализацией сразу нескольких ролей на MSC диаграммах.

   (b) Встречалась идея модификации предыдущего алгоритма с разрешением пропускать сообщения. Она была описана в [68], но широкого распространения не получила из-за невысокой достоверности результата.

   (c) Использование одного из алгоритмов [44], связанных с анализом внутренних состояний протокола, при этом параллельно выполняются две модели — MSC и SDL. В процессе выполнения проверяется, что принятые и посланные сообщения одинаковы.

[4] Имеется ввиду посимвольное совпадение сообщений трассы с сообщениями диаграммы.

Данные алгоритмы изначально разрабатывались для проверки непротиворечивости моделей, и перестают работать для проверки соответствия SDL диаграмм MSC при наличии разногласий между MSC и SDL, типа отсутствия цикличности на MSC документации и ее присутствия на SDL модели.

Условия построения CASE системы

Сформулируем условия, которым должна соотвествовать CASE система, в которой применяются и MSC, и SDL модели, на основе интеграции имеющихся подходов и собственного опыта разработок. Должны присутствовать следующие возможности:

1. использовать SDL модель независимо от MSC модели. Например, если некоторый стандарт задан в виде SDL диаграмм, то должна быть возможность использовать готовую SDL модель, без того, чтобы начинать строить MSC модель;

2. автоматически строить SDL модель по MSC модели. Это существенно ускоряет скорость разработки;

3. проверять согласованность описаний на MSC и на SDL, причем с учетом изменений, появляющихся в течение жизненного цикла программы.

Проблемные места

При этом следует выделить следующие проблемные места, на которые следует обратить внимание.

- Разработать процедуру верификации[5] так, чтобы она позволяла более надежно проверять соответствие моделей при их рассогласовании, что неизбежно в течение жизненного цикла. Существующие подходы по верификации являются достаточно “хрупкими”, наличие даже

[5]Данный термин используем в качестве сокращения фразы “проверка соотвествия”.

незначительных изменений приводят к тому, что модели считаются несогласованными. Проверка должна быть более гибкой и учитывать возможность незначительного различия моделей, вызванного различными целями MSC и SDL моделей.

- Разработать процедуру генерации[6] SDL по MSC так, чтобы можно было получить ряд SDL моделей, подходящих к текущей ситуации, а не единственный вариант, выдаваемый автоматически.

- Существующие подходы по генерации и верификации базируются на неявном предположении, что есть MSC модель, которая эквивалентна SDL модели. На самом деле это не так, и стандарт MSC крайне неудобен для создания полного описания поведения, на нем можно описать лишь несложные варианты взаимодействия. Соответственно, язык описания MSC должен быть расширен, чтобы сделать возможным и генерацию, и верификацию.

Данные проблемы были успешно решены. Предложенные решения рассматриваются в данной работе. Их интеграция между собой и с технологией описана в работах [21, 20].

Работа с частями моделей, предназначенными для описания статических данных системы

Данная кандидатская работа нацелена на решение проблем в области стыковки динамик MSC и SDL моделей. То есть алгоритмов в области обмена сообщениями. Хотя данные модели имеют средства описания статических частей системы, но проблем в данной области гораздо меньше, поэтому данного момента мы лишь бегло коснемся. С точки зрения автора, в рамках технологии не может быть нескольких различных описаний статик, поэтому в рамках технологии REAL статическая часть системы — это отдельная самостоятельная модель. В последствии данным описанием пользуются как MSC, так и SDL модели. На рисунке 3 изображены различные

[6] Под генерацией понимаем создание одной модели по другой модели.

подходы по совместному использованию MSC и SDL в рамках технологии, а так же их интеграция с описанием статической части системы.

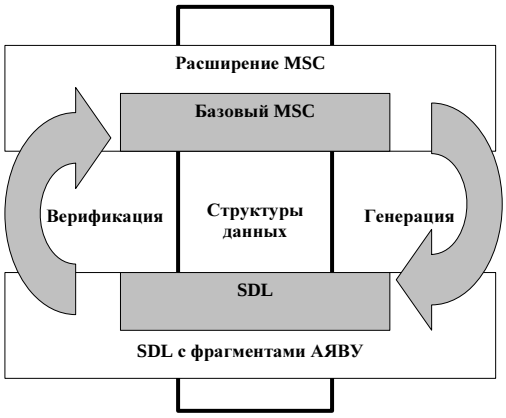

Рис. 3: Интеграция подходов в технологии REAL

Научная новизна

Научная новизна диссертационной работы заключается в следующем.

1. Разработана новая математическая модель верификации, которая работает в большем классе случаев, порождаемых ситуациями жизненного цикла программы, и позволяет проводить проверки, когда другие методы не могут работать.

2. Впервые поставлена задача настройки генерируемого SDL кода под требования технолога. Разработана оригинальная система генерации для построения SDL диаграмм на основе MSC модели, удобных для последующей работы с ними. Система позволяет интерактивно влиять на получаемый SDL код, так, чтобы обеспечить легкость его прочтения и понимания человеком в рамках текущей задачи. Она позволяет также провести оптимизацию уже имеющегося SDL кода.

3. Разработано новое расширение MSC, которое позволяет создавать описания реальных систем за счет большей описательной гибкости, чем в других моделях. Именно наличие данного расширения дает возможность использовать и генерацию, и верификацию, поскольку

позволяет эффективно описывать обратные ветви, и на основе этого создавать реальные системы.

4. Представлен оригинальный взгляд на совместное использование MSC и SDL моделей, использующий разработанные алгоритмы верификации, генерации, расширения MSC и сочетания статических данных MSC и SDL.

Данные решения были проинтегрированы с технологией и апробированы при создании телефонной станции "Юнивер–А" — многоканального радиоудлинителя телефонных линий. Результаты 1 и 2 применимы в ряде других систем, в которых система, заменяющая MSC, сводится к конечно-автоматному описанию. Результат 2 также применим для систем, если в них SDL модель является промежуточной.

Благодарности

Автор выражает признательность следующим людям, помогавшим ему при создании данной работы:

- Терехов Андрей Николаевич — за поддержку и веру в перспективность метода;
- Булычев Дмитрий Юрьевич — за жесткие требования к формализму и структуре моих работ;
- Захаров Михаил Николаевич — за помощь в получении статей;
- Иванов Александр Николаевич — за консультации по внутренним структурам и алгоритмам технологического средства REAL;
- Алексеева Светлана Леонидовна — за консультации по применению REAL'а в реальных задачах и за критику моей кандидатской работы;
- Шалупов Леонид Борисович — реализовавшего импорт текстовых SDL диаграмм в технологическое средство REAL.

А так же благодарит всех тех, кто участвовал с создании цепочки технологий RTST — RTST++ — REAL, на базе которых и были созданы данные решения.

Глава 1

Используемые формализмы

Для начала напомним существенные определения из теории конечных автоматов [1], на основе которых будем строить собственный аппарат. Иногда для конечных автоматов будем использовать краткую англоязычную аббревиатуру FSM (Finite State Machines).

Определение 1. *Будем называть конечным автоматом набор* $M = (Q, V, T, E, S, F)$.

- Q — *конечное множество состояний,*
- V — *конечный алфавит,*
- T — *множество правил перехода,* $T \subseteq Q \times V \times Q$,
- E — *множество* ε*-переходов,* $E \subseteq Q \times Q$,
- S — *начальное состояние,* $S \in Q$,
- F — *множество завершающих состояний,* $F \subseteq Q$.

Не умаляя общности, считаем, что множество пустых переходов доопределяется следующим образом:

- $\forall q \in Q : (q, q) \in E$,

- вместо E рассматривается E^* — для каждой вершины строится транзитивное замыкание ε-переходов.

В дальнейшем, вводя автомат M_i, будем подразумевать существование соответствующим образом проиндексированной шестерки $(Q_i, V_i, T_i, E_i, S_i, F_i)$.

Будем допускать вольность в использовании множеств T и E, считая, что существуют одноименные отображения $T : Q \times V \to 2^Q$ и $E : Q \to 2^Q$.

Определение 2. *Расширяем отношение/отображение* $T \subseteq Q \times V \times Q$ *до* $T^* \subseteq Q \times V^* \times Q$ *следующим образом:*

1. $\forall q \in Q : T^*(q, \varepsilon) = E(q)$,
2. $\forall a \in V, w \in V^*, q \in Q : T^*(q, wa) = E(T(T^*(q, w), a))$,

где V^* обозначает множество всех слов, состоящих из произвольных последовательностей над $(V \cup \varepsilon)$, с соответствующими правилами для пустого символа ε.

Определение 3. *Будем называть языком конечного автомата*

$$Lang(M) = (w \in V^* | \exists f \in F; (S, w, f) \in T^*)$$

Определение 4. *Будем писать, что два автомата изоморфны* $(M_1 \cong M_2)$ *тогда и только тогда, когда существует биекция* $g : Q_1 \to Q_2$ *такая, что выполняется:*

- $T_2 = \{(g(p), a, g(q)) | (p, a, q) \in T_1\}$,
- $E_2 = \{(g(p), g(q)) | (p, q) \in E_1\}$,
- $S_2 = g(S_1)$,
- $F_2 = \{g(f)| \in F_1\}$.

Определение 5. *Автомат является детерминированным тогда и только тогда, когда:*

- $E = \{(q, q) | q \in Q\}$,
- $\forall q \in Q, \forall a \in V$ *множество* $T(q, a)$ *содержит не более одного элемента.*

Для любого автомата можно построить соответствующий ему детерминированный, порождающий тот же язык [15]. Соответствующую операцию над автоматом будем изображать как $det(M)$.

Отметим, что если мы:

1. вычислим количество маркированных переходов в автомате;
2. заменим каждый маркированный переход на его номер;
3. проведем операцию детерминизации для получившегося автомата;
4. обратно заменим все номера на маркированные переходы,

то в результате получим автомат, в котором будут удалены все ε-переходы, но расположение сообщений не поменяется. ε-переходы исчезнут после операции детерминизации, а последняя операция их не добавит. Будем это использовать в главе “Генерация”.

Определение 6. *Детерминированный автомат* M *является минимальным, когда любой другой детерминированный автомат, порождающий* $Lang(M)$*, имеет не большее количество элементов в его множестве состояний.*

Для любого детерминированного автомата можно построить соответствующий ему минимизированный [11]. Данную операцию над детерминированным автоматом будем изображать как $min(M)$. Для произвольного автомата будем писать $min(det(M))$. Равенство языков автоматов означает изоморфность автоматов после операции $min(det(M))$, что дает отношение эквивалентности [15]. Соответствующие автоматы будем называть эквивалентными.

1.1 Определения, специфичные для части “Верификация”

Введем ряд определений, которые будут нужны для описания методов верификации.

Определение 7. *Автомат называется S-корректным, если*

$$\forall q \in Q \ \exists w \in V^* : q \in T^*(S, w)$$

Определение 8. *Автомат называется V-корректным, если*

$$\forall v \in V \ \exists (q_1, v, q_2) \in T$$

В рамках верификации будем рассматривать только V-корректные автоматы. Преобразование автомата к V-корректному очевидно.

Определение 9. *Пусть есть два автомата M_1 и M_2, для них выполнено условие $Q_1 \cap Q_2 = \emptyset$, тогда автоматом $M_3 = M_1 \oplus M_2$, порождающим объединение языков автоматов M_1 и M_2, будем считать*

$$(\{q\} \cup Q_1 \cup Q_2, V_1 \cup V_2, T_1 \cup T_2, E_1 \cup E_2 \cup \{(q, S_1), (q, S_2)\}, q, F_1 \cup F_2)$$

При этом $q \notin Q_1 \cup Q_2$.

Условие $Q_1 \cap Q_2 = \emptyset$ накладывается лишь для конструктивности определения.

Определение 10. *Автоматом M, усеченным по алфавиту $W \subset V$, назовем автомат $M_1 = M|_w$. При этом:*

- $Q_1 = Q$,
- $V_1 = W$,
- $S_1 = S$,
- $F_1 = F$,

- $T_1 = \{(q_1, a, q_2) | (q_1, a, q_2) \in T, a \in W\}$,
- $E_1 = E \cup \{(q_1, q_2) | (q_1, a, q_2) \in T, a \notin W\}$.

Определение 11. *Будем говорить, что S-корректный автомат M_1 сильно вкладывается в M_2 начиная с S_2, если*

$$Lang(Q_1, V_1, T_1, E_1, S_1, Q_1) \subseteq Lang(Q_2, V_2, T_2, E_2, S_2, Q_2)$$

Введенное отношение вложения не зависит от F_1, F_2. В контексте задачи верификации это связано с тем, что в автоматах потенциально существуют состояния, в которые можно попасть из начального состояния, но из них нельзя попасть в завершающее состояние. Согласно определению, пути до них не порождают слов автомата. Маркируя все состояния автомата в завершающие, мы насильно добавляем в язык цепочки, порождаемые подобными путями.

Данное определение означает, что при последовательных переходах из начального состояния автомата M_1 мы можем всегда совершить парный переход в автомате M_2, промаркированный тем же символом, что и в автомате M_1.

Мы потребовали S-корректности автомата M_1. Это означает, что при рассмотрении переходов этого автомата у нас не будет недостижимых вершин. То есть не рассматриваем конструкцию из множества вершин и отдельной начальной — эта конструкция будет вкладываться куда угодно, но смысла в этом не будет.

Определение 12. *Будем говорить, что S-корректный автомат M_1 слабо вкладывается в M_2, если существует $S^* \in Q_2$ такое, что M_1 сильно вкладывается в $(Q_2, V_2, T_2, E_2, S^*, F_2)$.*

Поскольку предыдущее отношение не зависело от F_1, F_2, то данное тоже от них не зависит. Также в данном отношении нет зависимости от S_2. Фактически, мы поочередно выбираем состояния автомата M_2 в качестве начального и проверяем, будет ли выполняться требование сильной вкладываемости начиная с данного начального состояния.

Определение 13. *Будем говорить, что M_1 вкладывается в M_2, если:*

- M_1 — *S-корректный автомат,*
- $M_1|_{V_1 \cap V_2}$ *слабо вкладывается в* $M_2|_{V_1 \cap V_2}$.

В данном отношении нет зависимости от F_1, F_2, S_2; очевидно, что автомат $(Q_1, V_1, T_1, E_1, S_1, Q_1)$ можно заменить на эквивалентный с тем же алфавитом.

Теорема 1.

$$Lang(M_1) \subseteq Lang(M_2) \iff min(det(M_1 \oplus M_2)) \cong min(det(M_2))$$

Доказательство.

$$(Lang(M_1) \subseteq Lang(M_2)) \iff$$

$$(Lang(M_1) \cup Lang(M_2) = Lang(M_2)) \iff$$

$$(Lang(M_1 \oplus M_2) = Lang(M_2)) \iff$$

$$(Lang(min(det(M_1 \oplus M_2))) = Lang(min(det(M_2)))) \iff$$

$$(min(det(M_1 \oplus M_2)) \cong min(det(M_2)))$$

□

1.2 Определения, специфичные для предложенного расширения MSC

Введем ряд определений, специфичных для расширенной MSC модели.

Определение 14. *Будем называть недетерминированный автомат нормализованным, если:*

- $\forall q \in Q, \forall v \in V : (q, v, S) \notin T$,
- $\forall q \in Q, \forall v \in V : (S, v, q) \notin T$,

- $\forall q \in Q, q \neq S : (q, S) \notin E$,
- $\forall f \in F, \forall q \in Q, \forall v \in V : (f, v, q) \notin T$,
- $\forall f \in F, \forall q \in Q, \forall v \in V : (q, v, f) \notin T$,
- $\forall f \in F, \forall q \in Q, q \neq f : (f, q) \notin E$.

Назовем нормализацией операцию, которая приводит конечный автомат к нормализованному виду путем добавления дополнительных S и F состояний и созданием переходов на них из оригинальных S и F состояний.

Определение 15. *Будем называть блоком набор* $B = (Q, V, T, E, S, F, C, Z, K, R)$.

- (Q, V, T, E, S, F) — *нормализованный конечный автомат,*
- C — *множество вершин, маркированных как выход с номером,* $C \subset Q$,
- Z — *множество вершин, маркированных как выход без номера,* $Z \subset Q$,
- K — *вершина, маркированная как соединение вниз,* $K \in Q$,
- *пересечение любых двух множеств из набора* $\{S\}, F, C, Z, \{K\}$ *пусто,*
- R — *функция, при которой всем состояниям автомата из* C *сопоставлены натуральные числа,* $R = \{(q, n) | n \in \mathbb{N}, q \in C\}$,
- $\forall r \in C \bigcup Z \bigcup \{K\}, \forall q \in Q, \forall v \in V : (r, v, q) \notin T$,
- $\forall r \in C \bigcup Z \bigcup \{K\}, \forall q \in Q, \forall v \in V : (q, v, r) \notin T$,
- $\forall r \in C \bigcup Z \bigcup \{K\}, \forall q \in Q, q \neq r : (r, q) \notin E$,
- $\forall q \in Q$ *существует не более одного элемента* $(q, r) \in E$, *где* $r \in C \bigcup Z \bigcup \{K\}$.

Так же, как и в случае с автоматами, будем подразумевать существование соответствующим образом проиндексированного набора $(Q_i, V_i, T_i, E_i, S_i, F_i, C_i, Z_i, K_i, R_i)$ для блока B_i.

Определение 16. *Частные случаи:*

- *блок является корректным, когда* $C = \emptyset$ *или* $Z = \emptyset$ *или* $C \bigcup Z = \emptyset$*;*
- *блок является простым, когда* $C = \emptyset$ *и* $Z = \emptyset$*;*
- *процедурой является блок* $C = \emptyset$*;*
- *функцией является блок* $C \neq \emptyset$*,* $Z = \emptyset$ *и* K *недостижимо из* S*.*

Отметим отношения между определениями:

- любой простой блок $B = (Q, V, T, E, S, F, \emptyset, \emptyset, K, \emptyset)$ можем считать конечным автоматом, ограничив его набором $M = (Q, V, T, E, S, F)$.
- по любой процедуре $B = (Q, V, T, E, S, F, \emptyset, Z, K, \emptyset)$ можем построить простой блок $B = (Q, V, T, E_1, S, F, \emptyset, \emptyset, K, \emptyset)$, где $E_1 = E \cup \{(r, K) : r \in Z\}$. Поэтому считаем, что простой блок и процедура — это одно и то же.

Определение 17. *Пусть для блоков* B *и* B_1 *выполнено* $Q \cup Q_1 = \emptyset$*, тогда операцией конкатенации по* K *для блоков* B *и* B_1 *назовем операцию, выдающую блок* $B_2 = \bigotimes_K(B, B_1)$ *со следующими свойствами:*

- $Q_2 = Q \bigcup Q_1$,
- $V_2 = V \bigcup V_1$,
- $T_2 = T \bigcup T_1$,
- $E_2 = E \bigcup E_1 \bigcup \{(K, S_1)\}$,
- $S_2 = S$,
- $F_2 = F \bigcup F_1$,

- $C_1 = C \bigcup C_1$,
- $Z_1 = Z \bigcup Z_1$,
- $K_2 = K_1$,
- $R_2 = R \bigcup R_1$.

Расширяем данную операцию на случай, когда
$(Q \setminus (\{S, K\} \cup C \cup Z \cup F)) \bigcap (Q_1 \setminus (\{S_1, K_1\} \cup C_1 \cup Z_1 \cup F_1)) \neq \emptyset$.
В этом случае:

1. для всех элементов q из этого пересечения заменим все его использования в B_1 на соответвующий ему $\tilde{q} \notin (Q \cup Q_1)$;
2. проводим операцию, поскольку необходимые условия выполнены;
3. в множество E_2 добавляем все пары соответствия $(q, \tilde{q})$ и $(\tilde{q}, q)$.

Определение 18. *Будем называть супеблоком уровня n набор* $L = (Q, V, T, E, S, F, C, Z, K^1, \ldots, K^n, R)$. *Его определение отличается от определения блока тем, что* K *состоит не из одного элемента, а из множества* $\{K^1, \ldots, K^n\}$. *Аналогично вводится корректность суперблока.*

Определение 19. *Пусть у нас задано натуральное число n, набор блоков* $B, B_1, \ldots, B_n$, *и разбиение* X, *для которых выполнено:*

- B *— функция,*
- *пересечение любых двух множеств из набора* $Q, Q_1, \ldots, Q_n$ *пусто,*
- X *разбивает* C *на набор множеств* $C^1, \ldots, C^n$
  - *не пересекающихся,*
  - *возможно пустых,*
  - *в объединении дающих все* C.

Тогда операцией конкатенации по X для блоков $B, B_1, \ldots, B_n$ назовем операцию, выдающую суперблок $\tilde{L} = \bigotimes_X(B, B_1, \ldots, B_n)$ со следующими свойствами:

- $\tilde{Q} = (\bigcup_{i=1}^{n} Q_i) \bigcup Q$,
- $\tilde{V} = (\bigcup_{i=1}^{n} V_i) \bigcup V$,
- $\tilde{T} = (\bigcup_{i=1}^{n} T_i) \bigcup T$,
- $\tilde{E} = (\bigcup_{i=1}^{n} E_i) \bigcup E \bigcup (\bigcup_{i=1}^{n} \{(p, S_i) : p \in C^i\})$,
- $\tilde{S} = S$,
- $\tilde{F} = (\bigcup_{i=1}^{n} F_i) \bigcup F$,
- $\tilde{C} = \bigcup_{i=1}^{n} C_i$,
- $\tilde{Z} = \bigcup_{i=1}^{n} Z_i$,
- *для всех i:* $\tilde{K^i} = K_i$,
- $\tilde{R} = \bigcup_{i=1}^{n} R_i$.

Аналогично операции конкатенации по K, расширяем данную операцию до случая, когда в каких-либо парах набора множеств $((Q \setminus (\{S, K\} \cup C \cup Z \cup F)), (Q_1 \setminus (\{S_1, K_1\} \cup C_1 \cup Z_1 \cup F_1)), \ldots)$ есть общие элементы.

- Для одинакового элемента q в паре множеств переобозначаем один из них в собственном блоке через соответствующий ему $\dot{q} \notin ((\bigcup_{i=1}^{n} Q_i) \bigcup Q)$. Таким образом, повторяя данную операцию некоторое количество раз, возможно, для одинаковых q в разных блоках, придем к тому, что одинаковых элементов больше не останется.
- Проведем операцию конкатенации по X для модифицированных блоков.
- Проведем преобразование, обратное первому шагу, добавляя в $\tilde{E}$ пары $(q, \dot{q})$ и $(\dot{q}, q)$ для всех проведенных операций замены.

Определение 20. *Пусть у нас задано натуральное число n и набор блоков $B_1, \ldots, B_n$, так, что пересечение любых двух множеств из набора $Q_1, \ldots, Q_n$ пусто. Назовем операцией конкатенации по A для блоков $B_1, \ldots, B_n$ операцию, выдающую блок $B = \bigotimes_A(B_1, \ldots, B_n)$ со следующими свойствами:*

- $Q = \bigcup_{i=1}^{n} Q_i$,
- $V = \bigcup_{i=1}^{n} V_i$,
- $T = \bigcup_{i=1}^{n} T_i$,
- $E = (\bigcup_{i=1}^{n} E_i) \bigcup (\bigcup_{i=1}^{n} \{(\dot{q}, S_i)\}) \bigcup (\bigcup_{i=1}^{n} \{(K_i, \ddot{q})\})$,
- $S = \dot{q}$,
- $F = \bigcup_{i=1}^{n} F_i$,
- $C = \bigcup_{i=1}^{n} C_i$,
- $Z = \bigcup_{i=1}^{n} Z_i$,
- $K = \ddot{q}$,
- $R = \bigcup_{i=1}^{n} R_i$.

Где $\{\dot{q}, \ddot{q}\} \bigcap (\bigcup_{i=1}^{n} Q_i) = \emptyset$.

Аналогично операции конкатенации по X, тем же алгоритмом расширяем данную операцию до случая, когда в каких-либо парах набора множеств $((Q_1 \setminus (\{S_1, K_1\} \cup C_1 \cup Z_1 \cup F_1)), \ldots, (Q_n \setminus (\{S_n, K_n\} \cup C_n \cup Z_n \cup F_n)))$ есть общие элементы.

Глава 2

Верификация SDL диаграмм по MSC диаграммам

MSC и SDL диаграммы описывают одно и то же — обмен сообщениями, только дают на него взглянуть с разных точек зрения. MSC диаграммы показывают картину с точки зрения стороннего наблюдателя. При этом видно поведение объектов в совокупности, но не ясно, по каким принципам каждый из объектов принял то или иное решение. SDL диаграммы, наоборот, изображают картину с точки зрения одного объекта, показывают, на основе чего он выбрал тот или иной вариант поведения, но рассматривается поведение только одного объекта и нет информации о поведении других объектов в этот же момент времени.

Когда в жизненном цикле возникают два типа независимых диаграмм, которые описывают одно и то же, их несогласованность может быть причиной многих ошибок. Поэтому появляется проблема сравнения двух моделей. Данная задача возникает как при начальном создании SDL и MSC диаграмм без процедуры генерации, так и в процессе разработки при изменениях моделей для того, чтобы знать, не появилось ли в них противоречий.

2.1 Существующие подходы

Рассмотрим задачу сравнения SDL реализации и MSC документации. Точного решения здесь предложить нельзя, поскольку, во-первых, SDL и MSC модели являются моделями разных формальных уровней. При этом то, что на MSC записано в виде комментариев на естественном языке, на SDL записано в виде работы с переменными. Во-вторых, возможность использования конструкций общего вида на SDL ведет к алгоритмической неразрешимости задачи.

При этом необходимость их сравнения важна, и обе модели сильно похожи. И та, и другая задают некоторую расширенную конечно-автоматную модель, алфавитом которой являются сообщения. По сути, надо сравнить порождаемые языки этих конечно-автоматных моделей и сформулировать результат проверки.

В существующих подходах для начала делается предположение, что доверие к MSC диаграммам больше. Они являются эталоном, а SDL диаграммы являются проверяемыми. Для проверок используются следующие подходы, которые условно можно разделить на три категории.

1. Из MSC диаграммы объекта генерируются трассы — линейные последовательности сообщений. После этого данные последовательности “накладываются” на SDL модель объекта, начиная с некоторой точки. При наложении сообщения должны быть одинаковыми и пропускать их нельзя. При наложении значения переменных, их анализ и операции с ними не учитываются. Поскольку на MSC можно описывать циклы, то потенциально набор трасс бесконечен. Мы ограничиваем себя в выборе конечного набора, например, по максимальной длине трассы, и проверяем лишь его. Если сумели наложить все эти трассы, то считается, что SDL модель объекта соответствует его MSC описанию. Если подобная проверка прошла успешно для всех объектов, то считается, что MSC описание системы соответствует SDL описанию системы.

2. Модификация предыдущего подхода, когда разрешено пропускать сообщения на проверяемой модели [68].

3. Параллельное исполнение SDL и MSC моделей как двух белых ящиков. При исполнении принимаемые и порождаемые цепочки сообщений должны быть одинаковыми. Каждая из моделей может состоять из набора автоматов, исполняемых параллельно [44]. В идеальном варианте параллельно исполняется сразу вся SDL система и вся MSC система. Использование полного выполнения систем или введение каких-либо ограничений зависит от объема системы и количества доступных ресурсов. В процессе выполнения либо обнаруживаются расхождения в цепочках сообщений, либо, с учетом поставленных ограничений на перебор, считается, что две системы соответствуют друг другу.

Более детально о философии проверки соответствия MSC и SDL диаграмм можно прочитать в стандартах серии ETSI [40, 39, 41], а о реализации подходов 1 и 3 — в документации на средство Telelogic Tau [72].

Подход 2 не получил широкого распространения из-за недостоверности результатов, а подходы 1 и 3 достаточно экстремистичны в своих требованиях. Например, условие на последовательное совпадение сообщений слишком сильное. Оно нарушается простым добавлением нескольких сообщений для накопления статистики еще одним объектом. Так же при создании систем его можно нарушить еще десятком способов. За всем этим можно следить, параллельно редактируя не одну, а сразу обе модели, но в реальных проектах SDL и MSC, без описанного в данной работе расширения MSC, часто так и остаются только похожими, но не эквивалентными.

2.2 Постановка задачи

Пусть в процессе разработки системы мы получили MSC и SDL модели. Полного соответствия между ними нет. Причиной этого является трудность создания и поддержания одинакового поведения сразу и на MSC, и на SDL без нововведений, предложенных в данной работе. Про модели доподлинно

известно, что они *описывают одну систему* с возможными "незначительными" изменениями, такими как:

- на MSC при описании циклично выполняющегося сценария цикличность не описывается;
- реализации на SDL модели нескольких MSC ролей;
- введение дополнительных сообщений на SDL модели;
- существование "дополнительных" объектов на SDL, отсутствующих на MSC диаграммах;
- разбиение MSC роли на несколько сервисов;
- отсутствие объединения различных MSC сценариев одного объекта в единое целое;
- отсутствие на MSC диаграммах символа завершения существования объекта.

Ставится задача проанализировать MSC документацию и SDL реализацию именно в таком виде с выявлением возможных ошибок.

Утверждается, что подобная ситуация появляется достаточно часто в жизненном цикле систем, и при этом существующие алгоритмы проверки моделей на соответствие оказываются малоэффективными из-за того, что практически всегда SDL модель "не совпадает" с MSC моделью. Автор предлагает "зайти с другого конца" и проверять, что в SDL модели нет противоречий с MSC документацией. Таким образом, моделям разрешено быть различными, но не должно быть ситуаций, когда они друг другу противоречат. Разумеется, предлагаемая проверка может быть использована вместе с уже существующими методами, описанными в предыдущем разделе.

При построении алгоритма используются следующие соображения:

1. мы считаем MSC диаграммы априори достоверными, а SDL диаграммы — проверяемыми;

2. мы отказались от анализа переменных на SDL и комментариев на MSC. В качестве информации, взятой с моделей, используем конечно-автоматные модели, описывающие языки, алфавитом которых являются сообщения;

3. мы проверяем, что у SDL реализации есть хотя бы одна возможность правильно обработать MSC спецификацию. Это означает, что если у системы есть два варианта выполнения, один из которых ведет к ошибке, а другой — к положительному результату, то выбираем "положительный". Например, когда сообщение нужно и может придти вовремя, а может придти заранее и уничтожиться из-за несохранения в состоянии, то выберем случай, когда оно придет вовремя;

4. считаем ошибкой ситуацию, когда существует трасса с MSC спецификации, которая не имеет отображения в SDL реализацию. Как бы мы корректно ни исполняли программу, все равно она не соответствует спецификации;

5. возможно, что реально программа из-за ошибок будет выполнять лишь часть всех вариантов поведения. В данном подходе рассматриваем только принципиальную возможность выполнения с точки зрения ограничений, введенных в пункте 2.

Таким образом, из SDL и MSC моделей мы извлекаем конечно-автоматную структуру. В следующем разделе опишем, на основе чего принимается решение о противоречивости/непротиворечивости данных конечно-автоматных структур. После этого будет дано объяснение смысла полученного результата в терминах исходных моделей. Если конечно-автоматные структуры друг другу не противоречат, то же заключение делаем и для моделей; если противоречат, то считается, что модели также друг другу противоречат.

2.3 Алгоритм верификации

На основе Определения 13 будем проверять соответствие SDL реализации MSC документации. При этом будем пытаться вкладывать автомат, полученный из MSC диаграмм, в автомат, полученный из SDL диаграмм. Условие на V-корректность выполняется по построению, поскольку в алфавит добавляем только те символы, которые встречаются на диаграммах. Также считаем, что условие на S-корректность для MSC также выполняется по алгоритму построения автомата. Недостижимые участки мы выбрасываем сразу. От SDL автомата мы S-корректности не требуем и это позволяет осуществлять проверку даже при незаконченности SDL диаграмм, когда не все вершины достижимы из начальной диаграммы, но переходы после SDL состояний являются корректными.

2.3.1 Обоснование метода в терминах исходной задачи

Разберемся с самым главным — а что подобная проверка дает с точки зрения исходной задачи? Какими свойствами обладает введенное отношение вложения автоматов в предметной области MSC и SDL диаграмм?

Для начала будем использовать следующее приближенное определение ошибки. Если на MSC диаграммах объекта была специфицирована цепочка **abc**, и если на SDL диаграмме этого же объекта используются символы из алфавита **{a,b,c}**, но нет цепочки $\alpha\mathbf{a}\beta\mathbf{b}\gamma\mathbf{c}\delta$, то это ошибка. При этом $\alpha, \beta, \gamma, \delta$ — некоторые цепочки над алфавитом SDL. Поэтому считаем, что такая цепочка обязательно должна существовать.

При выполнении операции усечения автоматов потенциально данная цепочка может преобразоваться в цепочку $\tilde{\alpha}\mathbf{a}\tilde{\beta}\mathbf{b}\tilde{\gamma}\mathbf{c}\tilde{\delta}$. Пусть в контексте задачи символы **a,b,c** обозначают “Вопрос”, “Ответ” и “Подтверждение”. Если мы допустим вставку внутрь цепочки **abc** каких-либо символов из алфавита **{a,b,c}**, то это существенно изменит ее смысл, и эта цепочка к первоначальной **abc** будет имет слабое отношение. Поэтому мы делаем предположение, что в рамках предметной области в результате операции усечения мы получим цепочку вида $\tilde{\alpha}\mathbf{abc}\tilde{\delta}$, т.е. такую, где в цепочку **abc**

уже ничего не будет вставлено. Это означает, что в автомате из SDL диаграммы после операции фильтрации будет существовать цепочка **abc**. При этом она может не начинаться в стартовом состоянии и не заканчиваться в завершающем. Она даже может не являться частью какого-либо слова из языка автомата SDL, например, из-за отсутствия в нем завершающих состояний. Но она обязательно существует. Рассмотрением бóльших цепочек получаем, что в контексте предметной области операция усечения имеет смысл и после выбора определенного алфавита сообщений "чистит" автоматы от сообщений, не входящих в данный алфавит так, что поиск цепочек становится более простым. Здесь имеется в виду, что после операции усечения символы цепочки уже идут последовательно.

При реализации и построении алфавитов обоих автоматов для каждого сообщения верен лишь один из трех вариантов.

1. Сообщение принадлежит и SDL, и MSC диаграмме. Значение такого сообщения для наших целей уже было разобрано выше. В рассмотренном примере это сообщения из алфавита {**a,b,c**}.

2. Сообщение принадлежит только алфавиту SDL диаграммы. Здесь надо выдать предупреждающее сообщение, свидетельствующее о том, что SDL реализует какую-то дополнительную функциональность, не специфицированную на MSC.

3. Сообщение принадлежит только алфавиту MSC диаграммы. Это означает, что SDL реализация заведомо не может выполнить функциональность, специфицированную на MSC. Здесь также надо выдать соответствующее предупреждающее сообщение о возможной ошибке (хотя такая ситуация может быть допустимой, если одна MSC-роль была разбита на несколько SDL-сервисов).

Следующей операцией, которую производим в процессе проверки, является операция перемаркировки всех состояний автоматов в завершающие. Очевидно, что на возможность найти одну цепочку в другом автомате данная операция не влияет. Операция перемаркировки состояний является частью преобразования автоматов для того, чтобы не потерять никакие цепочки. Например, при создании MSC диаграмм основное внимание уделяется специфицированию обмена сообщениями. При этом символ завершения объекта часто опускается, а также может подразумеваться наличие некой функциональности после данного сценария. Это приводит к тому, что язык порождаемых автоматов может оказаться пустым из-за отсутствия символа завершения. А нам надо сохранить те цепочки, которые на нем специфицированы. Поэтому перевод всех состояний в завершающие — дань предметной области.

Обсудим еще один тонкий момент — переход от нахождения существования отдельных цепочек к вкладываемости автоматов. В терминах исходной задачи это выражено в том, что если на MSC определен некий связный блок поведения, то и на SDL ему должен соответствовать связный блок. В рамках рассматриваемого подхода считаем, что интуитивное понятие связности, если SDL реализация соответствует MSC документации, не должно изменяться. Рассмотрим гипотетический пример. Если на MSC диаграмме сначала идет общая преамбула, а затем есть четыре варианта поведения, то и на SDL диаграмме должно быть нечто, похожее на преамбулу и четыре варианта. А если там будет реализована преамбула, а затем будет лишь три варианта, а четвертый будет реализован где-то в стороне, то данный подход будет считать это несоответствием, хотя в этом примере цепочки, соответствующие всем четырем вариантам, будут успешно найдены. Поэтому требуем, чтобы найденные цепочки имели общее начало. Это реализуется поиском состояния, куда вкладывается целый автомат MSC диаграммы. Вкладываемость автомата, согласно данным определениям, соответствует вкладываемости языка, порождаемого автоматом, а это сразу обеспечивает то, что все цепочки языка будут иметь общее начало.

Согласно примечанию к определению вкладываемости 13, нет зависимости от начального состояния S_2. В качестве этого состояния практиче-

ски выступает образ стартового состояния с SDL диаграммы. В терминах исходной задачи это достаточно естественно. Искомая функциональность должна где-то существовать, но ее выполнение может не начинаться с самого начала SDL диаграммы — может не следовать сразу за стартовым состоянием.

Мы последовательно рассмотрели все операции, которые производим в процессе проверки вкладываемости одного автомата в другой. Подведем итоги обсуждения. Метод может осуществлять проверку в случаях, которые соответствуют естественному развитию системы в течение жизненного цикла. Приведем пример.

Пусть на этапе проектирования был создан ряд MSC диаграмм. Завершения функциональности объектов на них никто не указывал, поскольку планировалось, что на данной группе сценариев объект не заканчивает свое существование. Потом была построена SDL модель, реализующая данную функциональность. На SDL модель позже были добавлены дополнительные обмены сообщениями. Внутрь нее — сообщения для сценариев техобслуживания. Также сообщения были добавлены до и после отработки данной функциональности. Еще SDL модель была преобразована таким образом, что объект стал заканчивать свое существование после ее отработки. Над первоначально изображенной схемой на MSC диаграммах на SDL модели были проведены изменения, позволяющие ей выполняться циклично.

Все остальные методы проверки будут выдавать ошибку, поскольку SDL диаграмма была изменена слишком сильно относительно исходной MSC документации. Это естественно, так как соответствия здесь уже нет. Но можно проверить, что между MSC документацией и SDL реализацией нет противоречий. Это обеспечивается предлагаемым методом. Фактически, он может “найти” исходную модель внутри преобразованной при условии, что она удовлетворяет вышеприведенному набору предположений. В описанном случае он подтвердит, что противоречий нет. В другом случае, если как-либо изменить образ самой первой SDL модели, построенной по MSC в преобразованном SDL, то метод справедливо выявит ситуацию ошибки.

2.3.2 Реализация

Для начала полностью выпишем алгоритм, проверяющий соответствие SDL реализации MSC документации для одной сущности, а затем прокомментируем, на основе чего взялись те или иные пункты. В квадратных скобочках даны необязательные улучшения алгоритма, позволяющие увеличить его скорость работы.

1. По проверяемой паре MSC–SDL создаются конечные автоматы. Под MSC имеется в виду MSC диаграмма, выбранная в качестве начальной, и все те, которые оказались с ней связаны. См. раздел 2.3.3.

2. Все состояния автоматов перемаркируются в завершающие.

3. Анализируются списки сообщений, выдаются соответствующие предупреждения, выполняется процедура усечения автоматов по списку сообщений, принадлежащим обоим автоматам.

4. Запускается алгоритм проверки вложения усеченного автомата из MSC в усеченный автомат из SDL.

   (a) [Детерминизировать и минимизировать усеченный автомат из MSC.]
   Результат обозначим как $A - MSC$ [1].

   (b) По всем вершинам из усеченного SDL автомата.

      i. Считать начальной вершиной усеченного SDL автомата выбранную.

      ii. [Проверить возможность вложения $A - MSC$ по начальным символам слов в новый SDL автомат. Перейти к следующей вершине при определении невозможности вложения.]

      iii. Детерминизировать и минимизировать новый SDL автомат. Результат обозначим как $A - SDL$.

[1] Если операция детерминизации и минимизации не проводилась, то вместо данного автомата берем усеченный MSC автомат.

iv. Склеить автоматы $A - MSC$ с $A - SDL$ согласно Определению 9.

v. Детерминизировать и минимизировать автомат с предыдущего шага. Результат обозначим как $A - \sum$.

vi. Если $A - \sum$ изоморфен $A - SDL$, то внести данную вершину в множество вершин, начиная с которых усеченный MSC автомат вкладывается в усеченный SDL автомат.

(c) Выдать вершины, начиная с которых усеченный автомат из MSC вкладывается в усеченный автомат из SDL.

Определения вложения были даны неконструктивно, но приведенная теорема позволяет построить конструктивный алгоритм проверки вложения. Сделаем несколько замечаний по дополнительным пунктам, которые мы ввели:

1. усеченный автомат из MSC диаграмм можно минимизировать, поскольку от замены данного автомата на эквивалентный порождаемый язык не меняется. Это дает выигрыш в последующих шагах алгоритма, сложность которых зависит от количества вершин;

2. следует более осторожно пользоваться алгоритмами $min(det(\dots))$, имеющими большую сложность. Требуется проверить, что $Lang(M_1) \subseteq Lang(M_2)$ после того, как все состояния были перемаркированы в завершающие. Пусть известно, что слова длины n образуют множества P_1 и P_2 соответственно, и для них известно, что $P_1 \not\subset P_2$. Тогда из этого сразу следует, что $Lang(M_1) \not\subset Lang(M_2)$. А n можно взять равным 1 или 2.

Теперь о сложности используемых алгоритмов. Верхние границы временной сложности для алгоритмов по количеству вершин автоматов:

- детерминизации — экспоненциальный;
- минимизации — $O(n * ln(n))$;

- проверки на изоморфность — $O(n^2)$.

На практике верхние границы являются завышенными:

- для алгоритма детерминизации автоматы берутся из MSC и SDL диаграмм. Это означает, что соединений между вершинами мало и итеративный алгоритм детерминизации сходится быстро;
- от связки детерминизация-минимизация часто можно избавиться с помощью дополнительного пункта 2, проверяя только те вершины, для которых вложение может состояться;
- в алгоритме изоморфности автоматы могут отсеяться до основной проверки, например, по количеству вершин или по количеству вершин в множестве завершающих состояний.

Это и дает возможность практического применения. Дальнейшие возможности по увеличению быстродействия уже стоит обсуждать в контексте конкретной реализации конечных автоматов.

При проверке системы последовательно рассматриваем все возможные MSC–SDL пары. Для данных пар должно выполняться условие наличия на них общих сообщений. Таким образом производится проверка всей системы, а затем пользователь может просмотреть результаты анализа.

2.3.3 Выбор начальной MSC диаграммы

По множеству базовых MSC диаграмм для каждого объекта можно выделить, как его диаграммы связаны друг с другом с помощью reference и inline конструкций. Вся совокупность MSC диаграмм объекта образует несвязный ациклический направленный граф [9]. При построении конечного автомата надо указать, с какой диаграммы его надо начать строить. Автоматически определить, какая диаграмма является “начальной”, невозможно. Поэтому она задается. После этого на множестве MSC диаграмм организуется порядок, и мы получаем дерево, начинающееся с выбранной MSC диаграммы[2].

[2]Для HMSC [46] начальную диаграмму задавать не надо, поскольку соответствующая информация на них уже содержится. Метод применим и к MSC, и к HMSC без модификаций алгоритма.

В зависимости от наличия ошибок в соответствии между MSC и SDL диаграммами выбор начальной диаграммы может влиять на результат проверки:

- при "соответствии" SDL диаграмм множеству MSC диаграмм данного объекта метод будет давать положительные результаты для любой MSC диаграммы, выбранной в качестве начальной;
- при наличии ошибок в "соответствии" для каких-то диаграмм, выбранных в качестве начальной, ответ может быть положительным, а для каких-то — отрицательным.

В качестве пояснения можно рассмотреть пример, когда у нас есть три MSC диаграммы A, B, C. A порождает символ a и ссылается на диаграмму B, B порождает b и ссылается на C, которая порождает c. Если выберем в качестве начальной диаграммы A, то автомат будет порождать цепочку abc. Если B — bc, если C, то c. Далее надо рассмотреть SDL диаграмму, порождающую $abbc$.

При организации проверки всей системы метод позволяет рассчитать результаты проверки для всех MSC диаграмм в качестве начальной и выдать интерпретацию результата.

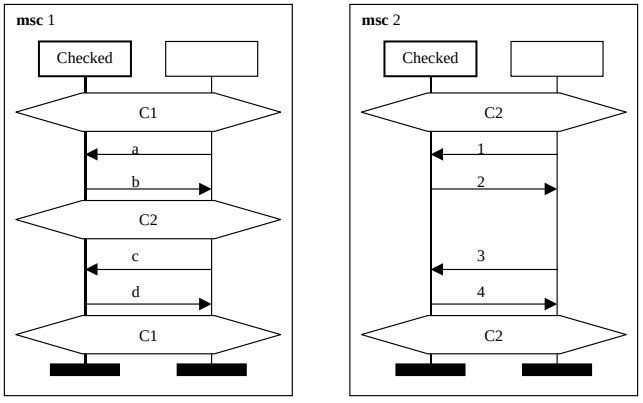

Рис. 2.1: MSC диаграммы 1, 2 для проверки соответствия

2.4 Пример

Пусть существуют MSC спецификации 1, 2, 3, 4. Спецификации 1 и 2 изображены на рисунке 2.1, а спецификации 3 и 4 — на рисунке 2.2. Пусть также дана SDL реализация, приведенная на рисунке 2.3. Ставится целью проанализировать соответствие реализации различным наборам спецификаций.

Метод позволяет выявить, что SDL реализация:

1. удовлетворяет MSC спецификации 1, без учета спецификаций 2, 3 и 4;
2. удовлетворяет MSC спецификации 2, без учета спецификаций 1, 3 и 4;
3. удовлетворяет спецификациям 1 и 2 вместе, если начальной является спецификация 1;
4. удовлетворяет спецификациям 1 и 2 вместе, если начальной является спецификация 2;
5. удовлетворяет спецификации 3, учитывая остальные спецификации;
6. не удовлетворяет спецификации 4, учитывая остальные спецификации.

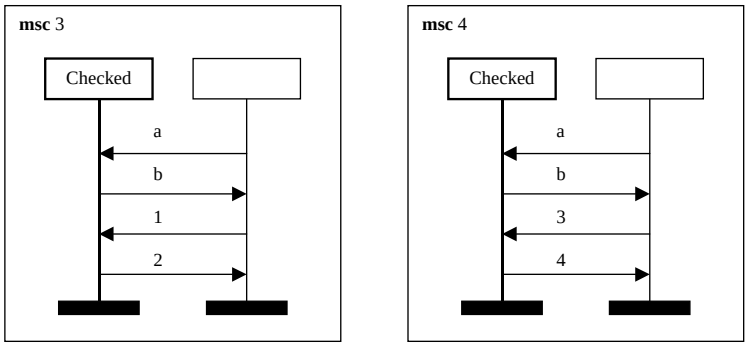

Рис. 2.2: MSC диаграммы 3, 4 для проверки соответствия

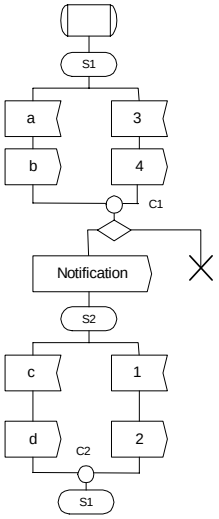

Рис. 2.3: Проверяемая SDL диаграмма

2.5 Обработка сложных случаев

В разделе 2.3 мы дали базовый алгоритм. В ряде случаев, вытекающих из специфики используемых стандартов, алгоритм должен быть расширен. Соответствующие ситуации будут описаны в данном разделе.

2.5.1 Проверка при наличии в SDL коде конструкции Save

Конструкция Save сильно затрудняет верификацию из-за влияния на обрабатываемый порядок сообщений. Соответствующий пример продемонстрирован на рисунке 2.4. Интуитивно приведенная реализация (SDL) не соответствует спецификации (MSC), но сообщения, поданные в порядке спецификации, будут обрабатываться реализацией. Разработчик CASE средства должен решить, разрешить ли подобный стиль программирования в своем средстве, или нет.

Поскольку в данной работе мы обсуждаем не методы программирования, а возможности по верификации диаграмм, то покажем, что предлагаемый метод также может быть применен и для оператора Save. Продемонстрируем необходимые преобразования на примере с рисунка 2.5. Пусть конструкция сохранения используется в одном состоянии. Справа

на рисунке 2.5 изображены некоторые[3] случаи, которые может обрабатывать эта конструкция и которые существенны для демонстрации данной проблемы:

1. BXAY — нормальное выполнение, левая ветвь;
2. ABXY — сохранение сообщения, затем использование сохраненного сообщения;
3. BXDZ — нормальное выполнение, правая ветвь;
4. ABXDZ — сохранение сообщения, затем уничтожение его без использования.

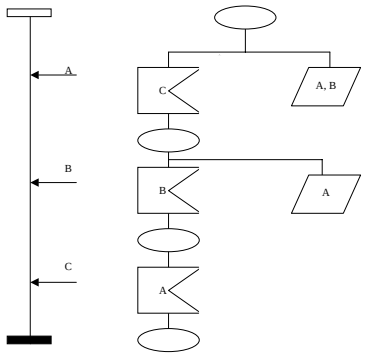

Рис. 2.4: Пример возможного несоответствия SDL кода MSC документации из-за использования конструкции Save

Одна из возможных реализаций SDL кода — это заведение отдельной очереди [47], в которую складываются сохраненные сообщения. При данной реализации автомат, у которого есть сохраненное сообщение в очереди, отличается от автомата, у которого сохраненного сообщения нет. Вышеописанная проблема решается аналогичным способом. Создается несколько копий автомата, которые индексируются сохраненными сообщениями. Копии автомата модифицируются, между копиями создаются переходы,

[3]Реально случаев больше:

- многократные приходы сохраняемого сообщения;
- приход и уничтожение сообщения (см. раздел 2.5.2).

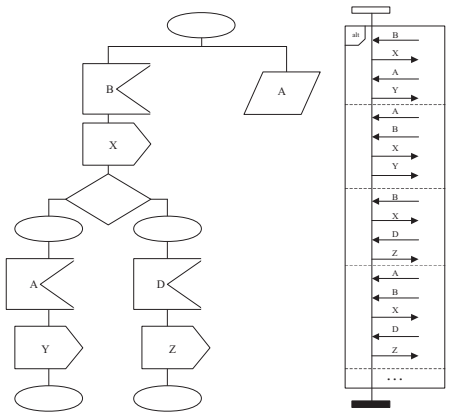

Рис. 2.5: Пример с конструкцией Save (SDL) и варианты его исполнения (MSC)

которые соответствуют добавлению и выниманию сохраняемого сообщения в/из данной очереди. Изменение копии автомата (изменение состояния очереди) происходит в момент, соответствующий выходу из состояния. Очередь моделируется с заданной глубиной. Автомат, проиндексированный двумя сохраненными сообщениями A, отличается от автомата с одним сохраненным сообщением A. Случай с очередью нулевой глубины соответствует случаю, когда оператор сохранения сообщения ставится только для обработки ситуаций, когда сообщение приходит раньше времени, а использование очереди как специального аккумулятора запрещено.

Схематично продемонстрируем создание переходов для вышерассмотренной конструкции с очередью длины 1 на рисунке 2.6. Между автоматами были добавлены следующие переходы:

1. переход с записью сохраняемого сообщения A в специальную очередь и переход на копию автомата, проиндексированную этим состоянием очереди;

2. переход обратно, когда используется сохраненное сообщение;

3. переход обратно, когда сообщение уничтожается без использования.

Созданная модель из нескольких копий автомата моделирует использование конструкции Save, что можно проверить по рассмотренному примеру.

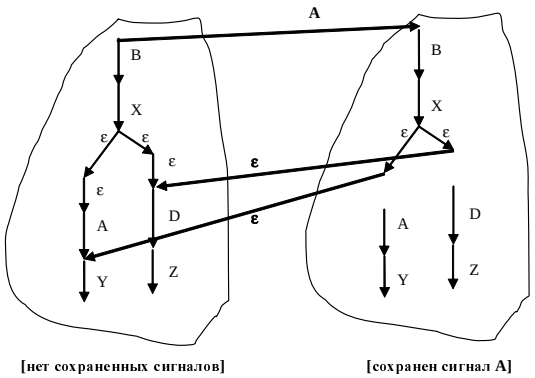

Рис. 2.6: Построение автомата для конструкции Save

2.5.2 Уничтожение сообщений, не обрабатываемых в текущей ситуации

В соответствии со строгой спецификацией SDL, сообщения, которые не обрабатываются в данном состоянии и не сохраняются, должны быть уничтожены. Это означает, что реализация (SDL) с рисунка 2.7 может обработать заданную там же спецификацию (MSC). Разработчик своего CASE средства должен выбрать, соответствует ли подобная ситуация здравому смыслу или нет. Данный выбор делается на основе предметной области и реализации SDL модели в конкретной операционной системе. Наилучшим выбором является предоставление пользователю дополнительных настроек о допустимости подобных конструкций.

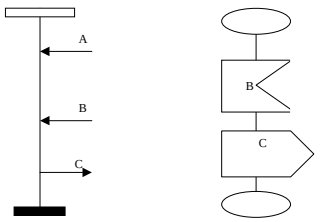

Рис. 2.7: Возможное несоответствие кода документации из-за уничтожения необрабатываемых сообщений

Рассматриваемый метод способен верифицировать и эти случаи тоже. Для этого конечный автомат SDL расширяется командами по уничтожению лишних сообщений как это изображено на рисунке 2.8.

Рис. 2.8: Преобразования автомата для уничтожения сообщений

При этом:

1. на реальном конечном автомате $\bar{B}$ заменяется на множество всех входящих сообщений, кроме B;

2. если проблема рассматривается в комплексе с конструкцией Save, то дополнительно следует рассмотреть работу с очередью, ведущую к созданию переходов между копиями автоматов.

2.5.3 На MSC диаграммах присутствует оператор параллельного исполнения

В случае, когда на MSC диаграммах присутствует оператор параллельного исполнения, то предлагаемый метод применим тогда, когда выполняется дополнительное условие, что каждый из параллельных участков не содержит сообщений, принадлежащих другим MSC диаграммам данного объекта. После этого выполняется серия проверок. Каждую из конструкций параллельного исполнения заменяем на выбранный из нее единственный сценарий. Рассматривая все такие конструкции, получаем произведение возможных ситуаций. Считается, что SDL реализация соответствует MSC документации, если все конечные автоматы всех вариантов с MSC диаграмм вкладываются в SDL реализацию.

Рассмотрим ситуацию с рисунка 2.9. Данную MSC диаграмму разбиваем на случаи $A - B - D$ и $A - C - D$, и требуем, чтобы каждый из соответствующих им конечных автоматов укладывался на SDL реализацию. При проведении процедуры верификации при выполнении операции усечения автоматов удаление параллельных сценариев приводит к удалению соответствующей функциональности с SDL реализации. Согласно допол-

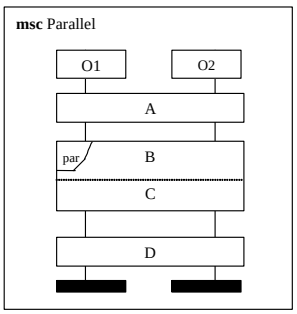

Рис. 2.9: Пример MSC диаграммы с параллельным исполнением участков

нительному условию данное удаление не затрагивает остальную функциональность. Далее пользуемся тем, что в усеченном SDL оставшийся из параллельных сценариев должен быть реализован.

2.6 Возможные усовершенствования и дальнейшее развитие метода

В данной работе описаны базовые идеи по проверке соответствия SDL реализации MSC документации. В дальнейшем метод планируется расширять. Приведем некоторые идеи, которые могут как сделать его более удобным в применении, так и перерасти в отдельную область по проверке соответствия SDL и MSC моделей, базирующуюся на конечных автоматах. Ожидается, что данная область будет дополнять существующие подходы на основе трасс и состояний протокола.

1. Разрешить пользователю редактировать словари сообщений, по которым сравниваются MSC и SDL диаграммы.

2. Предоставлять возможность задавать множество MSC диаграмм, используемых для проверки. При этом можно будет исключить из проверки диаграммы, являющиеся источником проблем.

3. На текущий момент данный метод выдает достаточно жесткое решение — либо да, либо нет. В текущей реализации пользователю приходится самостоятельно классическими средствами анализа трасс искать, где находится точка расхождения. Данный процесс можно авто-

матизировать для нахождения ближайшей точки расхождения диаграмм.

4. С помощью конечно-автоматных моделей можно также определить, насколько SDL реализация отличается от MSC спецификации. Это можно сделать на основе построения разности языков. Разность конечных автоматов из MSC и SDL диаграмм будет давать автомат, порождающий разность языков. При необходимости на основе него можно как построить наборы слов данного языка, соответствующие трассам сообщений, так и представить его в виде SDL диаграмм с помощью методов, используемых в части "Генерация". Особую ценность это имеет, когда ясно, что SDL реализация соответствует MSC спецификации, но не ясно, а что же она делает еще. Ни одно из известных автору средств не предоставляет подобной информации. В качестве аннотации существующих средств использовалась работа из области, близкой к данной [66].

5. Идею предыдущего пункта можно развернуть и предоставлять информацию о покрытии MSC спецификациями SDL реализации. Этого тоже нет ни в одном средстве, и наиболее ценно применение данного пункта — в тестировании. Отличие от покрытия с помощью трасс в том, что здесь можно будет задавать "тесты" не в виде конечного числа событий (трассы), а в виде сценариев с циклами. Если предыдущий пункт следует применять, когда существует единое описание объекта на MSC диаграммах, то данный пункт применим, когда есть много разрозненных MSC диаграмм, описывающих части SDL поведения.

2.7 Место подхода среди существующих методов

С точки зрения автора, круг подходов по верификации SDL диаграмм по MSC диаграммам практически всегда ограничивается рассмотрением

трасс, с возможным приложением к тестированию, и рассмотрением состояний протокола [44]. Даже в стандартах серии ETSI [40, 39, 41] упомянуты именно эти два подхода. Таким образом, средство Telelogic Tau [72] действительно соответствует стандартам и поддерживает все существующие подходы.

В работе [68] упоминалось о возможности пропускать сообщения на проверяемой модели. Но поскольку в ней рассматриваются только отдельные трассы, то достоверность такого подхода невелика. Автору не известны средства, реализующие подобный подход. В качестве аннотации существующих средств так же использовалась работа из области, близкой к данной [66].

В смежной области формальных методов существует отдельный подход по сравнению конечно-автоматных моделей на основе вложения языков. Хорошее описание формальных методов можно найти в книге NASA [43]. Использование конечно-автоматных моделей без словарей сообщений [53] будет похоже на алгоритмы [44] с той только разницей, что используются не переборные методы, а формальные. Соответственно, их использование будет иметь те же недостатки, что и средство Telelogic Tau.

Предлагаемый подход выигрывает именно за счет интеграции нескольких идей:

- переход от SDL и MSC к конечно-автоматным моделям с учетом специфики SDL и MSC;
- операции фильтрации, позволяющей разделить общее множество сообщений на “используемые” и “неиспользуемые”;
- проверки на включение языков, реализуемой с помощью конечных автоматов.

Это дало возможность на базе технологии REAL создать модель [17], в рамках которой можно проверять непротиворечивость диаграмм даже тогда, когда известно, что диаграммы не полностью соответствуют друг другу.

Глава 3

Генерация SDL диаграмм по MSC диаграммам

В процессе практического использования технологии REAL стало понятно, что наибольшую трудность для технолога вызывает переход от сценариев поведения системы в целом к поведенческой модели каждого отдельного объекта, т.е. переход от MSC к SDL. На момент написания статьи [19] данный разрыв выглядел так, как изображено на рисунке 3.1.

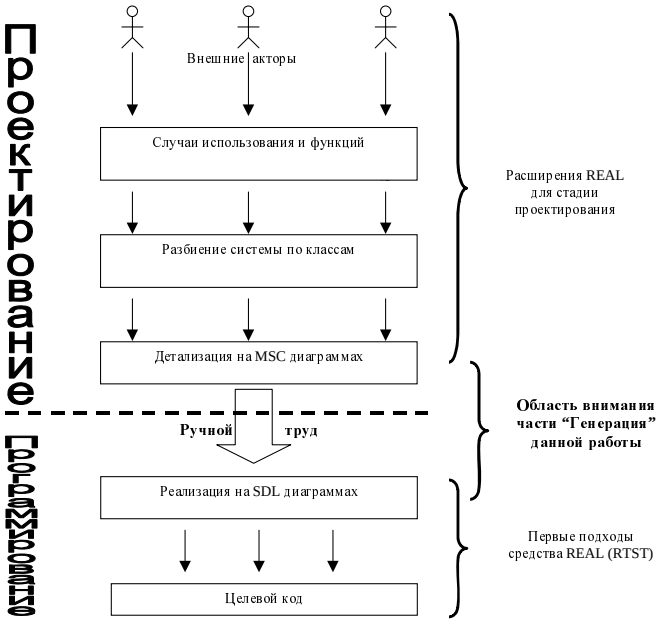

Рис. 3.1: Место SDL и MSC моделей в технологии REAL

Эта же проблема возникает и в других методологиях — как на базе UML 1.4 с Sequence и Collaboration диаграммами [64, 25], так и в подходах на базе MSC без UML, или в подходах на основе Use Case Maps [29]. В них также существует разрыв между MSC-подобной моделью и SDL.

Поскольку MSC и SDL диаграммы относятся к разным уровням создания системы (см. таблицу 1), то в стандартных средствах разработки сначала на этапе проектирования создается описание системы на MSC диаграммах, а затем осуществляется ручной переход от MSC диаграмм к SDL диаграммам. Таким образом, вся динамика системы разрабатывается дважды.

Подобная двойная работа совершенно избыточна, и возможен автоматический переход от MSC диаграмм к SDL диаграммам. Наличие подобного подхода позволяет:

- быстрее создавать системы;
- избегать многих ошибок при ручном переходе;
- быстро создавать прототипы систем и получать ряд оценок, в том числе и временных, будущей системы.

3.1 Текущее состояние проблемы

Разберем существующие подходы по переходу из MSC диаграмм в SDL диаграммы с различных точек зрения по их применимости в реальных задачах.

3.1.1 Однократный перенос

Для начала рассмотрим возможность однократного переноса информации с MSC диаграмм на SDL диаграммы. Автор выделяет две основных модели синтеза SDL диаграмм по MSC диаграммам[1]:

[1]Был проанализирован больший объем работ, но обычно сам алгоритм был закрыт, возможно по коммерческим соображениям, как в работах [73, 38]. Существует ряд алгоритмов [54, 51, 65, 71, 42, 70, 69, 56, 74, 67], посвященных переходам из MSC–подобных диаграмм в другие конечно-автоматные

1. группа работ, выполненных в канадском университете Конкордиа (Concordia University, Canada) [62, 30, 50, 35]. В дальнейшем данную модель синтеза будем называть “Канадской”;

2. серию московских работ [60, 58, 57, 13], выполненных в Институте Системного Программирования Российской Академии Наук, на основе которой было создано средство MOST [23], авторы которого принимали участие в создании средства KLOCWork [24, 22]. Позже данное средство было проинтегрировано в несколько измененном виде в средство Telelogic SDL Suite 4.4. Эту модель синтеза будем называть “Российской”.

Для того, чтобы показать сильные и слабые стороны возможности синтеза, произведем сравнение двух данных алгоритмов между собой по следующей шкале:

1. статическая часть SDL модели;

2. схема построения динамического поведения;

3. характеристики построенной динамической части SDL модели, по возможности их связи с алгоритмом построения.

Для алгоритмов работы “Канадской” модели характерно:

1. Сначала фиксируется SDL архитектура (разбиение на объекты, определение каналов связи между объектами, перечней сообщений каждого канала). Архитектура может быть задана средствами SDL модели [62] либо сгенерирована из UML [35]. После этого автоматическая генерация SDL диаграмм по MSC диаграммам осуществляется в уже зафиксированную архитектуру.

модели. К сожалению, рассмотренные в них алгоритмы малоприменимы для генерации SDL. SDL диаграммы имеют ряд ограничений по сравнению с другими автоматными моделями и, поэтому, достаточно сложно “втиснуть” получившуюся конечно-автоматную модель в SDL.

2. Схема построения динамического поведения[2]:

   (a) анализ возможности построения;

   (b) составление таблиц частичного упорядочивания сообщений с учетом архитектуры. При этом для каждого из каналов строится отношение предшествования между сообщениями либо говорится, что сообщения не находятся в подобном отношении;

   (c) выделение конечных автоматов из MSC;

   (d) построение SDL кода по автоматам с учетом таблиц частичного упорядочивания сообщений. Таблицы частичного упорядочивания используются для вычисления необходимого перечня сохраняемых сообщений конструкции Save.

3. В процессе синтеза динамической части SDL диаграмм вносимые изменения минимальны. Следствием этого является:

   - построенная SDL модель обладает “наглядностью”, это означает, что в полученной SDL модели легко узнается исходная MSC модель;
   - более корректное использование конструкции Save;
   - излишне громоздкая SDL модель даже там, где ее можно было бы оптимизировать;
   - для некоторых MSC моделей невозможно произвести синтез таким образом, хотя выполняющая ту же логику SDL модель существует и может быть построена другими алгоритмами синтеза (отсутствие детерминизации автомата).

Для алгоритмов “Российской” модели характерно:

1. По MSC модели синтезируется не только динамическое поведение, но и архитектура SDL модели.

[2]Алгоритмы статей приводятся в слегка модифицированном виде для облегчения их сравнения.

2. Схема построения динамического поведения:

   (a) выделение конечных автоматов из MSC;

   (b) детерминизация полученных автоматов;

   (c) минимизация автоматов;

   (d) построение SDL кода по автоматам.

3. При синтезе динамического поведения возможно автоматическое его преобразование. Как следствие:

   - динамика SDL модели может быть изменена относительно исходной MSC модели. Из-за этого SDL модель становится менее узнаваемой (процедуры детерминизации и минимизации);
   - более грубое использование конструкции Save, подразумевающее сохранение всех сообщений (трудности с "протаскиванием" точной информации через алгоритмы детерминизации и минимизации);
   - возможность синтеза даже в тех случаях, когда предыдущий алгоритм выдает невозможность реализации (использование детерминизации);
   - возможность построения более простых (оптимальных) SDL моделей в некоторых случаях, по сравнению с спроектированными MSC диаграммами (использование минимизации);
   - возможное ухудшение SDL модели в некоторых случаях (детерминизация и по выходным сообщениям, хотя это и не является необходимым для возможности сгенерировать SDL модель из конечного автомата).

3.1.2 Согласование изменяющихся диаграмм

Теперь перейдем к необходимости согласования MSC и SDL при изменениях, появляющихся в течение жизненного цикла.

В алгоритмах “Канадской” модели предлагается возможность изменения исходных MSC диаграмм с соответствующим инкрементальным изменением SDL диаграмм [50]. Это интересный подход, однако сами авторы упоминают, что он будет работать далеко не во всех случаях.

В алгоритмах “Российской” модели предлагается вообще не заниматься стыковкой MSC и SDL диаграмм, и попросту запрещено ручное развитие сгенерированных SDL диаграмм. После каждого внесенного на MSC диаграммы изменения необходимо заново сгенерировать SDL модель. Для этого в данных алгоритмах предлагается расширить MSC модель операциями с переменными, параметрами, условными состояниями, чтобы при генерации получать не шаблон SDL диаграмм, а готовую модель, дополненную кодом.

Это ущербный подход с точки зрения выбранной модели, поскольку MSC диаграммы допускают многократную прорисовку одного и того же участка на различных диаграммах, что ведет к необходимости “размножать” работу с переменными. Полный отказ от SDL модели является неудобным шагом для технолога, у которого стандарт на разрабатываемый продукт написан на языке SDL.

3.1.3 Обобщение результатов

Сразу отметим, что невозможность осуществить синтез во многих разумных ситуациях является очень слабым местом, поэтому сразу стоит отказаться от “Канадской” модели в пользу использования “Российского” алгоритма. Далее будем обсуждать именно его с точки зрения возможных улучшений.

1. В разделе про существующие решения уже упоминалось, что авторы “Российского” алгоритма расширили стандарт MSC для поддержки всего цикла разработки программ и запретили редактирование SDL диаграмм. С точки зрения автора это было не самое удачное решение и от него следует отказаться. Это означает, что снова получаем две модели — MSC и SDL. Тем самым обеспечиваем больший комфорт на стадии программирования за счет использования SDL модели, лучше

приспособленной для программирования. За это приходится платить необходимостью согласования моделей, и возникает процедура верификации.

2. И MSC, и SDL модели, в принципе, позволяют описывать структуру системы, но поскольку архитектура у системы едина, то мы вынесли всю работу со статикой в отдельный редактор на основе диаграммы классов, а все остальные диаграммы заимствуют информацию из созданных в нем моделей. Так мы избавились от согласования статических данных этих двух моделей.

3. С алгоритмом построения динамики сложнее. Цепочка детерминизация – минимизация обладает как рядом плюсов, так и рядом минусов. Еще раз поясним, что детерминизация необходима для того, чтобы расширить область применимости алгоритма, а минимизация необходима для нейтрализации побочных эффектов детерминизации, когда автомат мог получиться слишком большим. Соответствующий пример рассмотрен в разделе 3.4. Перечислим плюсы и минусы.

   Плюсы:

   - возможность синтеза даже в тех случаях, когда "Канадский" алгоритм выдает невозможность реализации;
   - возможность построения более оптимальных SDL моделей в некоторых случаях.

   Минусы:

   - динамика SDL модели может быть слишком сильно преобразована (процедуры минимизации и детерминизации преобразовали SDL код к виду, который удобен машине и не соответствует изначальным планам проектировщика);
   - возможное ухудшение SDL модели в части случаев (детерминизация и по выходным сообщениям может дать более сложную модель, чем это действительно надо).

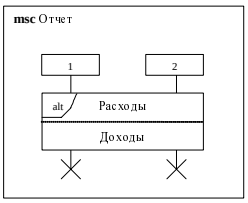

Рис. 3.2: Главная MSC диаграмма для генерации

Рассмотрим пример. Он связан с возможным ухудшением SDL модели при использовании данного алогритма. На рисунках 3.2, 3.3 показаны исходные MSC диаграммы, на рисунке 3.5 показана SDL модель, которая получена из оригинального алгоритма, на рисунке 3.6 показана наша модель. Обе модели были построены для объекта 1. Наша модель была получена с помощью детерминизации модели только по входным сообщениям. Автомат, из которого она была построена, не является детерминированным из-за наличия сообщения „Начало работы“. Но это сообщение является выходным, а, значит, подобная конструкция допустима. Модель, полученная с помощью “Российского” алгоритма, построена из автомата, детерминированного по всем сообщениям, в том числе и по выходным. Видно, что наша модель проще и по логике лучше соответствует исходным спецификациям.

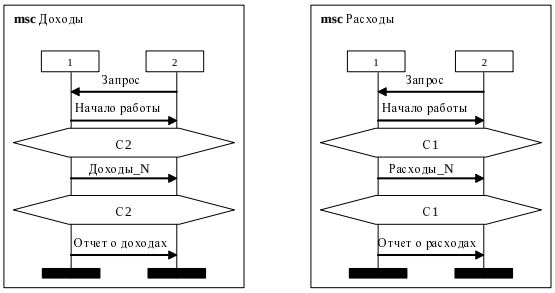

Рис. 3.3: Детализирующие MSC диаграммы для генерации

3.2 Предлагаемый подход

Наш подход базируется на идее, что конечная SDL модель предназначается для человека, поэтому неуправляемого алгоритма синтеза мало, проектировщик должен иметь возможность получить тот или иной SDL код по своему вкусу. Это означает, что некоторые участки кода могут быть “непра-

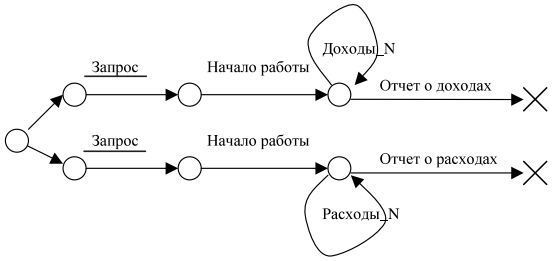

Рис. 3.4: Автомат, полученный из MSC диаграмм с рисунков 3.2 и 3.3

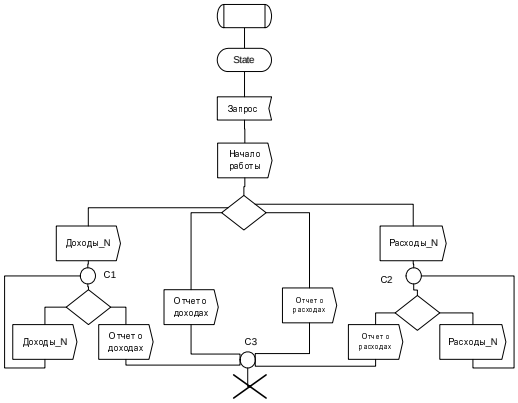

Рис. 3.5: „Классический“результат генерации

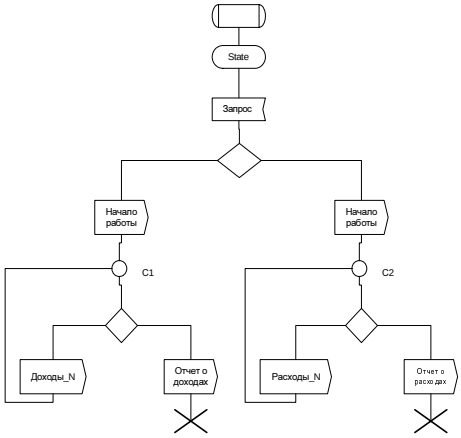

Рис. 3.6: Наш результат генерации

вильными", например, быть недетерминированными по выходным сообщениям либо быть неоптимальными с точки зрения процедуры минимизации, но удобными для последующей доработки.

Вспомним, что алгоритмы детерминизации были нужны для того, чтобы преобразовать конечный автомат к такому виду, из которого алгоритм синтеза мог бы построить SDL код. Однако любой автомат можно сначала проверить на возможность генерации из него SDL модели и при его корректности выполнить соответствующую генерацию.

Далее мы ввели промежуточную модель между MSC и SDL диаграммами для настройки процедуры синтеза. Это конечный автомат. С одной стороны, он уже несет информацию только о поведении данного объекта и игнорирует поведение других объектов, а, с другой стороны, он еще не нагружен специальной информацией о логике принятия решений и ограничениями на конечно–автоматную модель SDL. Из-за этого он более компактен, чем MSC и SDL модели и удобен для редактирования перед последующей генерацией SDL. В качестве примера можно взять рисунки 3.2, 3.3, 3.6, 3.4. Автомат редактируется человеком перед и после связки процедур детерминизация – минимизация:

Этим мы обеспечиваем:

- генерацию SDL кода в наиболее удобном виде;
- возможность защиты от процедур преобразования автомата (детерминизация и минимизация);
- исключение эффекта детерминизации по выходным сообщениям;
- выделение SDL процедур в конечном автомате.

В упомянутых подходах по автоматической генерации подобные возможности не могут быть поддержаны, поскольку оценить "читаемость" кода может только человек.

3.2.1 Статические данные

В отличие от идей по синтезу SDL архитектуры по MSC архитектуре [60], и синтеза необходимой динамики SDL [62] либо в готовую SDL архитектуру, либо в импортированную из UML [35], мы используем другой подход. Считаем, что у системы не может быть нескольких различных архитектур — будь то на UML, MSC либо на SDL. Поэтому мы выделили описание статики системы (разбиение на классы, наследование, агрегирование, описание структур обмена сообщениями, параметры сообщений, ...) в отдельный редактор, а уж SDL и MSC редакторы заимствуют информацию из него. Таким образом, статика системы у нас сразу согласована во всех диаграммах [19, 10].

3.2.2 Динамика

Для начала покажем построение базового алгоритма, а затем продемонстрируем, как его можно расширять для повышения читабельности получаемого SDL кода.

При построении SDL диаграмм для выбранного объекта проходим по следующей цепочке:

1. проверка корректности MSC описания. Соответствующие алгоритмы можно найти в работах [30, 34, 33];

2. переход от MSC диаграмм данного объекта к недетерминированному конечному автомату, порождающему все трассы данного объекта. (Построение срезов в терминах [13]);

3. опциональная защита автомата от процедур изменения;

4. детерминизация автомата;

5. минимизация автомата[3];

[3]Алгоритмам минимизации автоматов посвящены многочисленные работы, связанные с повышением быстродействия. Можно отметить, что генерация, по сравнению с верификацией, предложенной в данной работе, работает достаточно быстро и абсолютно не критично использовать алгоритмы минимизации со сложностью $O(n * ln(n))$, а достаточно $O(n^2)$.

6. “восстановление” автомата, если применялись алгоритмы пункта 3;

7. опциональное “переразложение” автомата и выделение процедур;

8. построение схемы SDL кода;

9. наращивание SDL работой с переменными и операторами.

Поскольку предложенные алгоритмы по генерации (как базовый, так и расширенный) SDL диаграмм из MSC диаграмм реально используют переход MSC→FSM→SDL, то вместо MSC диаграмм может быть задана любая модель, которая сводится к конечным автоматам. Это открывает возможность автоматического переноса информации из других технологий в SDL. Особо следует отметить, что исходной моделью может также служить и SDL модель. Таким образом, можем проводить оптимизацию уже написанного SDL кода.

В соответствии со сделанным замечанием будем считать, что перед синтезом SDL мы уже имеем конечный автомат. Условия на конечный автомат будут специфицированы далее.

3.3 Базовый алгоритм генерации

При построении алгоритмов будем идти от построения мелких деталей к построению глобального алгоритма. При этом в следующих разделах будет описано:

1. каким должен быть конечный автомат, чтобы появились те или иные SDL элементы;

2. как можно оптимизировать этот процесс, чтобы не порождать лишнее;

3. общая структура SDL кода, который хотим получить;

4. как породить некоторое дерево SDL кода;

5. как породить весь SDL код.

3.3.1 Требования по корректности автомата

Пусть у нас есть конечный автомат и дополнительная информация, указывающая тип сообщений. Какие из сообщений автомат принимает, а какие — посылает. Считаем, что на входе получаем корректный автомат, для которого будем рисовать основную картину расклейки, описанную дальше в разделе 3.3.3. Это означает, что имеем автомат:

1. без ε-переходов;
2. он детерминирован по принимаемым сообщениям. Это означает, что нет состояний, из которых выходят дуги, маркированные одним и тем же принимаемым сообщением, ведущие в разные состояния;
3. нет состояний, в которые нельзя попасть и которые не являются начальными;
4. нет состояний, из которых нельзя выйти и которые не являются завершающими.

Достаточно очевидно, что минимизированный детерминированный автомат этим условиям удовлетворяет.

3.3.2 Идеи порождения базисных SDL элементов

Опишем базовый набор правил для SDL элементов, которые будут порождаться из конечного автомата. Сразу заметим, что набор и расположение сообщений конечного автомата на этой стадии алгоритма меняться не будут, поэтому все преобразования будут происходить с генерируемым SDL.

1. Начальное состояние

   Начальное состояние автомата отличается от начального состояния SDL тем, что в него возможны переходы. Поэтому интерпретируем автомат как будто в него “добавили” еще одно состояние, создали из него ε-переход в начальное состояние, и дали ему статус начального состояния. “Предыдущее” начальное состояние после такого преобразования теряет свой особый статус и становится таким же, как и все остальные состояния.

2. Завершающее состояние

   Здесь есть аналогичное отличие. Из завершающего состояния автомата переходить можно. Поэтому интерпретируем автомат как будто был проделан ряд аналогичных “преобразований” по добавлению фиктивного завершающего состояния с ε-переходами в него из реальных завершающих состояний. После такого преобразования обработанные состояния становятся такими же, как и все.

3. Порождение SDL состояния

   После того, как с особыми состояниями мы уже разобрались с помощью преобразований 1 и 2, теперь остались лишь обычные состояния. Схематично изобразим такое состояние с принимаемыми и посылаемыми сообщениями, ведущими как в него, так и из него. Для SDL состояний (обычных) есть правило, что после них в обязательном порядке должен идти прием сообщений. Это правило является достаточным для выделения состояний конечного автомата, порождающих SDL состояния — см. рисунок 3.7. На левой части рисунка серым цветом изображена часть состояния автомата, порождающая состояние SDL. На этом же рисунке справа приведена аналогия, объясняющая различные типы сообщений и их направленность, если бы автомат строился по SDL. Все, что находится внутри пунктирной линии, может быть “стянуто” в состояние автомата. Прием и отправка сообщений будут заменены на маркированные дуги.

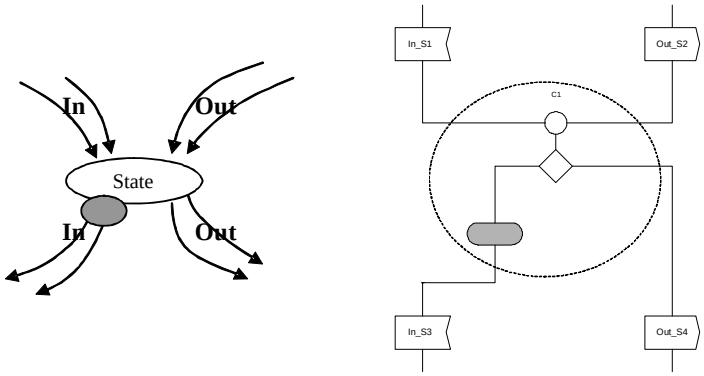

Рис. 3.7: Порождение SDL состояния (слева) и аналогия происхождения данного рисунка (справа)

4. Порождение соединителей

   Идея соединителей является достаточно простой. Есть необходимость из нескольких точек передавать управление в одну. В этом случае в месте передачи управления ставится ссылка на соединитель, а в том месте, куда передается управление, описывается соединитель и действия после него. При этом соединители являются нумерованными, что позволяет использовать на диаграмме объекта произвольный их набор.

5. Порождение операторов ветвления

   Операторы ветвления повторяют предыдущую конструкцию с точностью до наоборот. В некоторую точку передается управление, а затем есть несколько вариантов исполнения кода. Мы будем выявлять подобные ситуации и порождать требуемые операторы ветвления.

После того, как определили, откуда берется тот или иной элемент, покажем, каково их взаиморасположение при генерации.

3.3.3 Основная картина расклейки

Рисунок 3.8 является основным для базового алгоритма, и при описании алгоритмов будем все время на него ссылаться.

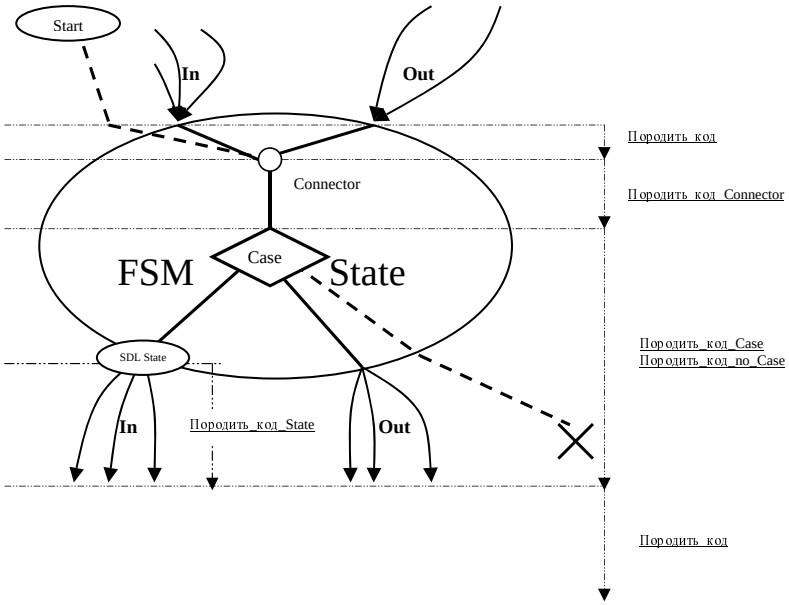

Рис. 3.8: Основная картина расклейки состояния автомата на SDL элементы и связь их с процедурами порождения кода

На рисунке 3.8 изображены:

1. состояние автомата (FSM State);
2. сообщения, изображенные в виде стрелок. Если стрелка направлена в данное состояние автомата, то для него сообщение считается входящим (графически расположены сверху от состояния). Иначе — выходящим (снизу). Стрелки несут следующую дополнительную информацию:
   - их тип помечен как **In** и **Out**. То есть, принимает объект данные сообщения, либо посылает;
   - графически показано, заходят они в состояние автомата или выходят. То есть, осуществляется операция с данными сообщениями до или после данного состояния автомата.
3. элементы SDL, которые порождаются наряду с посылкой и приемом сообщений:
   - SDL состояния:
     - начальное состояние (Start);

  - завершающее состояние (перекрещенные линии);
  - обычное состояние (SDL State).
- оператор вевления (Case);
- соединитель (Connector);
- соединительные линии.

Будем считать входами в состояние автомата:

- переход из “дополнительного состояния” Start;
- сообщения типа In, входящие в данное состояние;
- сообщения типа Out, входящие в данное состояние.

Графически все входы расположены над состоянием автомата.

Будем считать выходами из состояния автомата:

- переход в “дополнительное” завершающее состояние;
- выходящие In сообщения;
- выходящие Out сообщения.

Графически все выходы расположены под состоянием автомата.

Дополнительно к данному рисунку надо сделать ряд замечаний:

1. далеко не для всякого состояния порождаются все эти SDL элементы, и определим ряд процедур оптимизации, которые помогут выкинуть “лишние” элементы;
2. с помощью процедур порождения (участков) кода, изображенных справа от рисунка, определим алгоритмы построения SDL кода.

3.3.4 Необходимость появления SDL элементов из основной картины расклейки

Приведем набор правил, которые позволят не генерировать лишние элементы при порождении SDL кода по основной картине расклейки.

1. Если количество входов в состояние автомата равно одному, то порождение соединителя не нужно. Иначе нужно, за исключением пункта 6.

2. Подсчитаем количество переходов из состояния автомата. Из него могут быть следующие переходы:

   - переход в “дополнительное состояние” завершения;
   - выходящие In сообщения, порождающие один переход в порождаемое SDL состояние (один переход на все In сообщения);
   - выходящие Out сообщения, порождающие один переход на каждое сообщение (одинаковые сообщения порождают разные переходы).

   Если количество переходов из состояния равно одному, то порождение оператора ветвления не нужно. Иначе порождается оператор ветвления, при этом каждый из переходов порождает ветвь оператора ветвления.

3. Если в конечном автомате состояние помечено как начальное, то оно порождает специальное состояние Start. Иначе не порождает.

4. Если в конечном автомате состояние помечено как завершающее, то оно порождает переход в завершающее состояние. (Использований такого состояния, в отличие от предыдущего пункта, может быть больше одного.) Иначе не порождает.

5. Если у состояния есть выходящие In сообщения, то оно порождает SDL состояние. Иначе не порождает.

6. Если состояние порождает соединитель и SDL состояние, но не порождает оператор ветвления, то в этом случае соединитель не порождаются и все те линии кода, заходящие на соединитель, непосредственно соединяются с SDL состоянием.

Пусть в используемой реализации все состояния конечного автомата пронумерованы и, поэтому, удобно именовать порождаемые:

1. состояния как **State_<Номер_FSM_состояния>**;
2. соединители как **Connector_<Номер_FSM_состояния>**.

При подобном именовании элементарно может получиться, что **State_1** и **State_3** есть, а **State_2** не порождается по причине отсутствия выходящих In сообщений из FSM состояния 2. В дальнейшем будем считать, что состояния и их номера отождествлены, поэтому запись в угловых скобочках сразу обращается к номеру состояния: **State_<FSM_состояние>** и **Connector_<FSM_состояние>**.

Используя данные правила, по набору состояний сразу строится следующий набор массивов, состоящих из булевских величин, индексируемых состояниями:

1. порождающих начальное состояние (значение для состояния S обозначается как $StartState(S)$);
2. порождающих завершающее состояние ($Finish(S)$);
3. порождающих соединитель ($Connector(S)$);
4. порождающих Case (соответственно, его дополнение состоит из состояний, не порождающих Case) ($Case(S)$);
5. порождающих SDL состояние ($SDLState(S)$).

Если для всех состояний автомата построим основную картину расклейки, то будем иметь изображение, на котором, помимо всего прочего, изображен SDL код. Теперь надо его выделить в чистом виде.

3.3.5 Схема порождения описаний SDL элементов из основной картины расклейки

Определим, как будут порождаться SDL элементы в результате генерации в текстовый формат. Более детально он описан в Приложении А.

Последовательно пройдем по всем базовым элементам SDL и покажем схемы их построения:

- посылка сообщения Signal перейдет в **output Signal;**
- прием сообщения Signal перейдет в **input Signal;**
- переход в завершающее состояние в **stop;**
- переход в следующее обычное состояние State в **nextstate State;**
- переход на соединитель Connector в **join Connector;**
- описание оператора ветвления даст структуру вида:
  decision;
  decision_thread;
  дерево кода
  ...
  decision_thread;
  дерево кода
  enddecision;
- описание соединителя:
  connector Connector_...;
  дерево кода

- обычное состояние[4]:
  state State_...;
  input Signal_1;
  дерево кода
  ...
  input Signal_N;
  дерево кода

- начальное состояние:
  start;
  дерево кода

Где под деревом кода в графическом виде понимается объект, являющийся частью основной картины расклейки, который:

- имеет одну точку входа;
- имеет древообразную структуру;
- завершается либо переходом в SDL состояние (в обычное либо в завершающее), либо переходом на соединитель (переход включается, SDL состояние и соединитель — нет);
- при его обходе, помимо переходов предыдущего пункта, состоит лишь из операторов ветвления, дуг, маркированных выходящими сообщениями, и соединительных линий.

В качестве одной из идей для построения алгоритмов, отметим, что любое поддерево дерева само является деревом.

Дерево кода представимо в виде текстового описания. В этом случае оно представляет собой последовательность операторов, состоящих из посылки сообщения и оператора ветвления, заканчивающихся переходами на соединитель либо состояние. Построение текстового описания по конечному автомату определим в следующем разделе.

[4]Напомним о сделанном замечании для конструкции Save (См. Приложение А).

Отметим, что поскольку после SDL состояний и соединителей идут деревья, и существуют несложные алгоритмы представления деревьев в графическом виде, то это дает возможность построить также и графическое представление SDL.

3.3.6 Алгоритмы порождения частей дерева кода

После того, как определили схему порождения отдельного SDL элемента, разберемся, как можем породить этот элемент с соответствующим ему деревом кода.

Для этого рассмотрим процедуры, которые нарисованы на основной картине расклейки. Эти процедуры рекурсивно вызывают друг друга. Рекурсия завершается переходом либо на соединитель, либо в состояние[5]. При подобном обходе строится некоторое дерево SDL кода. Приводим алгоритмы на C–подобном псевдокоде.

Считаем, что по внутреннему представлению состояний автомата для каждого состояния S есть:

- ссылка на список выходящих сообщений, маркированных как In, обозначается как $S \rightarrow In\_Signal\_List$;
- ссылка на список выходящих сообщений, маркированных как Out, обозначается как $S \rightarrow Out\_Signal\_List$.

Согласно определениям корректности автомата, список выходящих сообщений, маркированных как In, не содержит повторений, а маркированных как Out, может содержать повторения.

Пусть по состоянию S и выходящему из него сообщению $Signal$ определяется состояние автомата, в которое совершается переход, если придет данное сообщение. Для сообщений, маркированных как In, это состояние

[5]То, что рекурсия обязательно закончится доказывается от противного.
В таком случае должна существовать бесконечная цепочка
единственное_сообщение_в_состояние→состояние
→выходящее_сообщение_из_состояния→...
Это противоречит корректности и конечности автомата. Сама на себя она зациклиться не может, поскольку вход в состояние в данной цепочке должен быть единственным.

определяется однозначно, а для сообщений, маркированных как Out, определяется номером сообщения в списке $S \rightarrow Out\_Signal\_List$. Соответствующую операцию будем обозначать как $NextState(S, Signal)$.

Теперь приведем алгоритмы процедур порождения.

1. Процедура порождает дерево кода, начиная с входа в состояние автомата.

   Породить_код(Состояние_S) {
   if(Connector(Состояние_S))
   Вывести "**join Connector_<Состояние_S>;** "
   else
   if(Case(Состояние_S))
   Породить_код_Case(Состояние_S)
   else
   Породить_код_no_Case(Состояние_S)
   }

2. Данная процедура порождает дерево кода для состояния автомата с части соединителя. Подразумевается, что соединитель действительно порождается.

   Породить_код_Connector(Состояние_S) {
   Вывести "**connector Connector_<Состояние_S>;** "
   if(Case(Состояние_S))
   Породить_код_Case(Состояние_S)
   else
   Породить_код_no_Case(Состояние_S)
   }

3. Данная процедура порождает дерево кода, для состояния автомата с Case части, подразумевается, что оператор ветвления действительно нужен.

Породить_код_Case(Состояние_S) {
  Вывести "**decision;**"
  if(Finish(Состояние_S)){
    Вывести "**decision_thread;**"
    Вывести "**stop;**"
  }
  if(SDLState(Состояние_S)) {
    Вывести "**decision_thread;**"
    Вывести "**nextstate State_<Состояние_S> ;**"
  }
  for(Signal in S→Out_Signal_List) {
    NS=NextState(Состояние_S,Signal);
    Вывести "**decision_thread;**"
    Вывести "**output Signal;**"
    Породить_код(NS)
  }
  Вывести "**enddecision;**"
}

4. Данная процедура порождает дерево кода, для состояния автомата с Case части. Подразумевается, что оператор ветвления в этом случае не нужен, ветвления нет, вариант один[6].

[6]Из-за требований на корректность автомата и обработки завершающего состояния один выход всегда есть. По алгоритмам для данной процедуры - только один.

Породить_код_no_Case(Состояние_S) {

if(Finish(Состояние_S)){

Вывести “**stop;**”

}

if(SDLState(Состояние_S)) {

Вывести “**nextstate State_<Состояние_S>;**”

}

for(Signal in S→Out_Signal_List) {

NS=NextState(Состояние_S,Signal);

Вывести “**output Signal;**”

Породить_код(NS)

}

}

5. Данная процедура порождает дерево кода, для состояния автомата с SDL State части. Подразумевается, что состояние действительно порождается.

Породить_код_State(Состояние_S) {

Вывести “**state State_<Состояние_S>;**”

for(Signal in S→In_Signal_List) {

NS=NextState(Состояние_S,Signal);

Вывести “**input Signal;**”

Породить_код(NS)

}

}

6. Данная процедура порождает дерево кода, для состояния автомата, являющегося начальным. Очевидно, что по основной картине расклейки, состояние *start* отделено от приема сообщений каким-либо другим SDL состоянием.

   Породить_код_Start_state(Состояние_S) {

   . . . *//Обработка критичных случаев с вырожденными автоматами*

   Вывести “**start;**”

   Породить_код(Состояние_S)

   }

3.3.7 Алгоритм порождения всего SDL кода

Теперь мы можем породить дерево кода, начиная с любого места состояния автомата. На основе введенных функций опишем, как мы можем получить весь SDL код по имеющемуся корректному автомату. Порождение дополнительных деталей, таких как описатели процесса, мы опускаем.

0. Построить массивы, в которых будет записано, какие элементы порождает каждое из состояний согласно алгоритмам раздела 3.3.5 ($StartState, Finish, Connector, Case, SDLState$).

1. Породить SDL код для начального состояния SDL

   for(State in StartState)

   Породить_код_Start_state(State)

2. Породить SDL код для всех состояний автомата, порождающих SDL состояние

   for(State in SDLState)

   Породить_код_State(State)

3. Породить SDL код для всех соединителей

   for(State in Connector)

   Породить_код_Connector(State)

3.4 Пример

Продемонстрируем выполнение метода на примере и параллельно поясним, зачем в алгоритме присутствуют те или иные части.

3.4.1 MSC диаграммы

На рисунках 3.9, 3.10, 3.11 изображен пример взаимодействия двух объектов, описанный на MSC диаграммах. Начальной диаграммой является диаграмма Main.

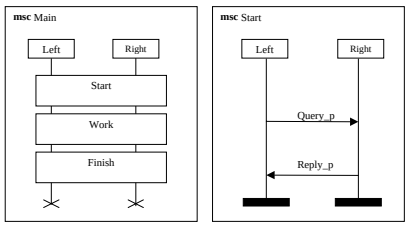

Рис. 3.9: MSC диаграммы Main и Start

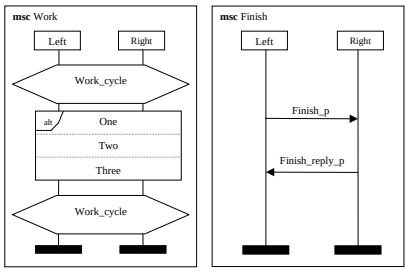

Рис. 3.10: MSC диаграммы Work и Finish

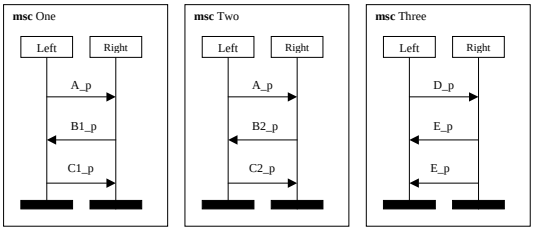

Рис. 3.11: MSC диаграммы One, Two и Three

3.4.2 Недетерминированный автомат

На рисунке 3.12 изображен недетерминированный автомат, полученный из MSC диаграмм. Видно, что даже при подправленной структуре автомата он сильно напоминает множество MSC диаграмм, из которого был сгенерирован[7]. Так же следует заметить, что он является недетерминированным. Для объекта Right он является недетерминированным по входящим сообщениям.

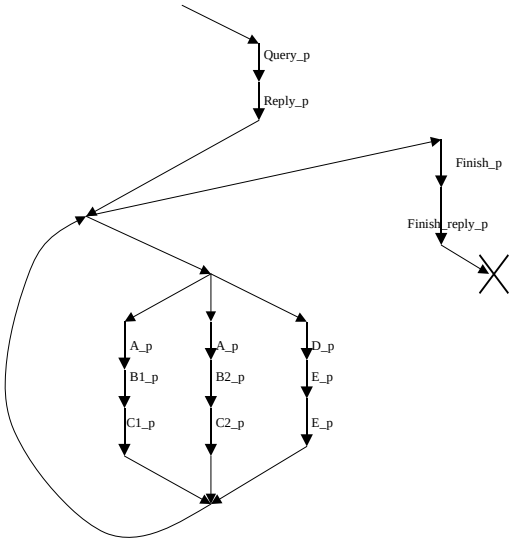

Рис. 3.12: Извлеченный автомат из MSC диаграмм

Рисунок 3.12 дает идеи о переходе от MSC к конечному автомату. Достаточно перейти к произвольному автомату, поскольку существуют алгоритмы по переходу от него к нужному виду автоматов.

3.4.3 Автомат без ε-переходов

Данный этап не является необходимым, существуют алгоритмы, которые сразу позволяют получить детерминированный автомат, но визуально данная структура, изображенная на рисунке 3.13, является достаточно удобной для восприятия, поскольку из нее выкинута лишняя информация, связанная с ε-переходами. Непосредственно по нему строить SDL код нельзя,

[7]Часто автомат после непосредственного перехода из MSC получается достаточно “дырявым” — в нем очень много лишних ε-переходов. Если демонстрировать его пользователю, то имеет смысл его немного оптимизировать. Например — заменять два последовательных ε-перехода в один.

поскольку он может оказаться недетерминированным по входным сообщениям, что противоречит требованиям по использованию алгоритма.

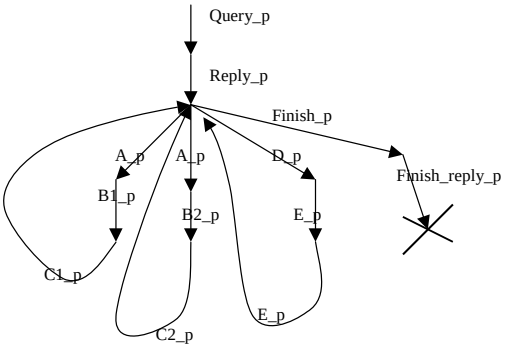

Рис. 3.13: Автомат без ε-переходов

3.4.4 Детерминированный автомат

Алгоритм детерминирования сделал свое дело, и в результате мы получили детерминированный автомат — рисунок 3.14. Данный автомат является излишне громоздким, поскольку достаточно было "склеить" переходы, маркированные как A_p на рисунке 3.13. Однако в реальном применении подобное "неоправданное усложнение" автомата действительно может встречаться. Значит, необходим еще следующий этап — минимизация автомата. Хотя и по данному автомату уже можно построить SDL код.

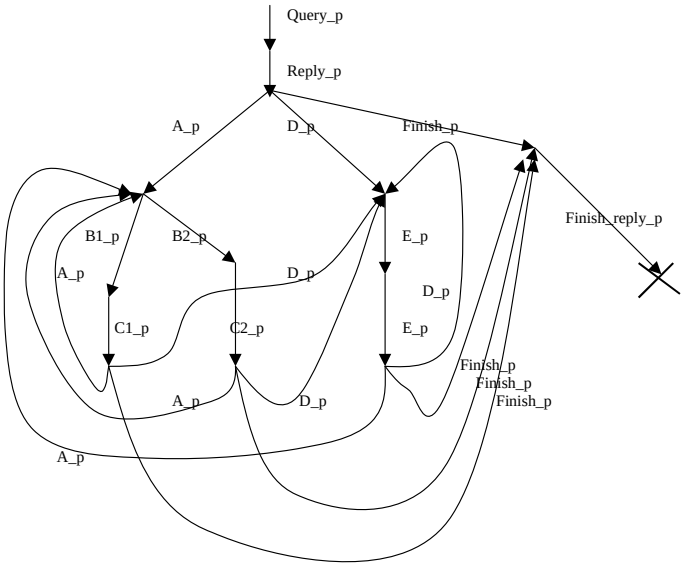

Рис. 3.14: Детерминированный автомат

3.4.5 Детерминированный и минимизированный автомат

Построенный минимальный вариант автомата изображен на рисунке 3.15. Он удовлетворяет требованиям корректности, поэтому можно применять процедуры построения SDL кода.

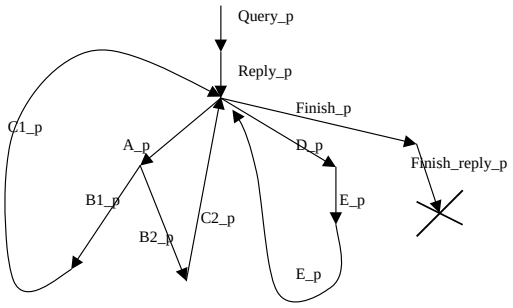

Рис. 3.15: Детерминированный и минимизированный автомат

3.4.6 Построенные SDL диаграммы

На полученный конечный автомат был запущен алгоритм генерации и он выдал следующий SDL код[8]. При этом входящие сообщения для одного автомата являются выходящими для другого. Это и дало различные структуры в SDL коде. На рисунке 3.16 изображен созданный SDL код для объекта Left. На рисунке 3.17 изображен созданный SDL код для объекта Right.

3.5 Сравнение с ручным программированием и оптимизация имеющегося SDL кода

Для того, чтобы оценить достоинства метода, проводились тесты по сравнению автоматической генерации SDL кода с SDL кодом, написанным человеком. Данные сравнения проводились по следующей схеме:

SDL→FSM→SDL

[8]Опять же, напомним о сделанном замечании для конструкции Save (См. Приложение А).

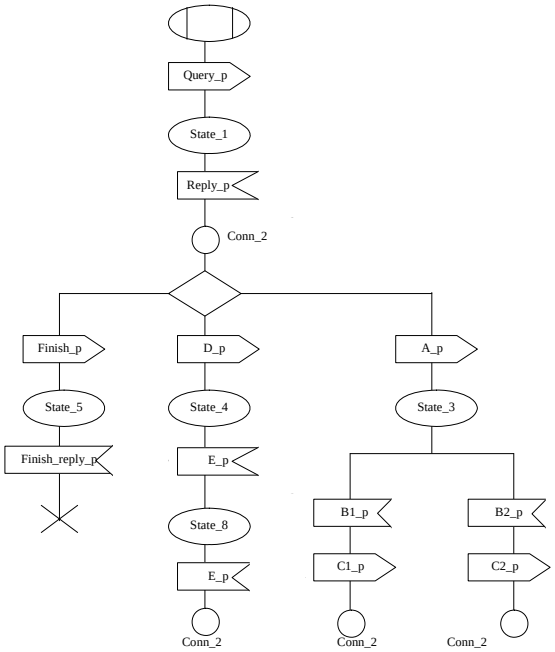

Рис. 3.16: SDL код для объекта Left

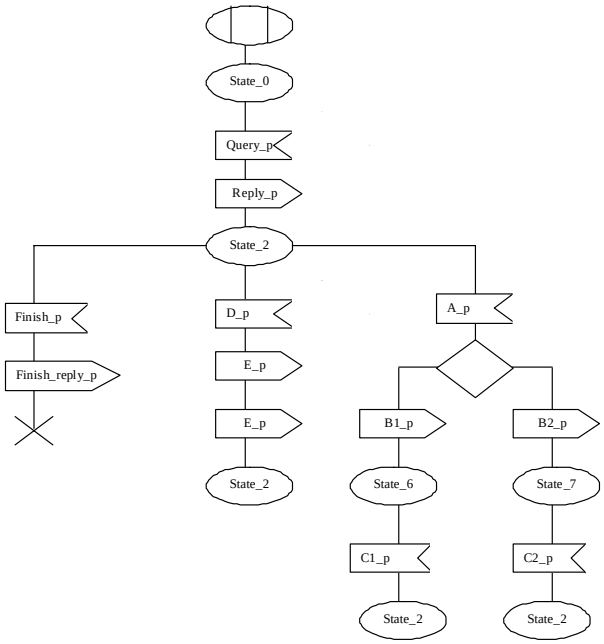

Рис. 3.17: SDL код для объекта Right

При этом брался уже написанный SDL код, затем по нему строился минимизированный детерминированный конечный автомат (при этом информация о структуре динамики SDL терялась полностью), а затем по данному автомату снова строился SDL код. После этого проводилось сравнение SDL кода человека и машины. Были получены следующие результаты.

1. Для автоматов с небольшим количеством состояний, средним количеством сообщений (примерно 10 состояний, 10 сообщений) и малым количеством соединительных линий код, написанный человеком и созданный машиной, оказывался одинаковым, с точностью до переименования состояний и соединителей.

2. На примерах, когда использовался автомат с бо́льши́м количеством соединений между состояниями и малым количеством сообщений (2-3 сообщения), код, написанный человеком, оказывался лучше. Данный эффект наблюдается из-за того, что при построении детерминированного автомата проводилась детерминизация и по входным сообщениям, и по выходным сообщениям.

3. На автоматах, когда присутствовало достаточно большое количество сообщений и состояний (больше 15) с малым количеством соединений между элементами, то автоматическая генерация кода является более продуктивной. В таких случаях человек создает код, который содержит избыточные элементы. Код, создаваемый машиной, лучше.

Еще раз сделаем акцент на методе, который был применен в данном разделе. Это переход из SDL диаграмм к конечному автомату и восстановление SDL диаграмм. Если исходный SDL код многократно переделывался и дописывался в течение жизненного цикла программы, то он становится неоптимальным. Подобный переход позволяет оптимизировать его событийную часть. Это означает, что последовательности сообщений обрабатываются, а логика работы с переменными теряется.

Суммарно отметим, что использование автоматической генерации, даже на основе базового алгоритма, позволяет резко сократить время разработки системы.

Практика использования системы синтеза показала, что подобное автоматизированное порождение SDL кода иногда не совпадает с человеческой логикой, и были проведены соответствующие исследования по улучшению читабельности SDL кода. Результаты этих исследований и решение проблемы детерминизации по входящим сообщениям изложены ниже. Данное улучшение применимо как к построению SDL диаграмм по MSC, так и к построению по SDL.

3.6 Улучшения базового алгоритма

Опыт работы с диаграммами, полученными при синтезе, показал, что описанный выше алгоритм является лишь первым приближением. Если SDL диаграммы предназначаются для последующего использования человеком (а не для прогонки на модели для получения временных оценок), то необходимо иметь средства управления автоматическим алгоритмом синтеза. Автор считает, что нужно перейти к автоматизированному процессу, когда пользователю дается возможность влиять на работу алгоритма для получения SDL кода более близкого к человеческой логике. В данном разделе показаны возможные улучшения базового алгоритма.

3.6.1 Краткий обзор базового алгоритма

При генерации для каждого объекта рассматривается следующая цепочка:
[MSC→] недетерминированный автомат→
недетерминированный автомат с удаленными ε-переходами→
детерминированный автомат→
минимизированный детерминированый автомат→SDL.

Недетерминированный автомат порождает язык, состоящий из всевозможных трасс данного объекта. Основной информацией о сообщениях является информация о “направлении” сообщения. Интуитивно — входящие сообщения порождают SDL состояния при генерации, а выходящие — нет.

3.6.2 Косметические улучшения

Можно настроить, в каком порядке от оператора ветвления будут появляться переходы в завершающее состояние, в обычные состояния, посылка сообщений. В приведенном алгоритме они появляются именно в данном порядке.

3.6.3 Порождение таймеров

Операции с таймерами достаточно просто “протягиваются” через основной алгоритм, поскольку все операции с таймерами данного объекта можно рассматривать как посылку соответствующего сообщения (запуск, остановка, перезапуск) объекту, маркированному идентификатором данного таймера. При построении SDL по MSC диаграммам сообщение от таймера рассматривается так же, как и любое другое входящее сообщение. Символы запуска и остановки таймера при построении конечного автомата порождают специальные маркированные дуги, которые в рамках алгоритма генерации нужно рассматривать так же как и посылку сообщений. При генерации кода эта “посылка сообщений” переходит в нужную операцию с таймерами с соответствующими установками.

3.6.4 “Макроопределения”

Под макроопределениями имеем ввиду:

1. “текущее состояние” “–”;
2. описатель “*” для входящих сообщений;
3. описатель “*” для состояний.

Можно проанализировать полученный SDL код и уменьшить его объем, используя вышеуказанные “макроопределения”. В ряде реальных ситуаций подобный подход может существенно уменьшить объем SDL кода.

3.6.5 Оптимальность автомата; “расщепление” деревьев

Поскольку одним из этапов построения SDL кода является минимизация автомата, то при данном построении производится попытка максимального уменьшения числа состояний автомата. Это ведет к тому, что то, что человек мог бы запрограммировать как две различных цепочки в двух деревьях SDL кода:
Посылка_x → Посылка_b → Посылка_c → State
и
Посылка_y → Посылка_b → Посылка_c → State
То при автоматическом построении будет ужато в:
Посылка_x → Connector
Посылка_y → Connector
Connector: Посылка_b → Посылка_c → State

Эта же ситуация может проявляться и на гораздо больших участках SDL кода, что схематически изображено на рисунке 3.18.

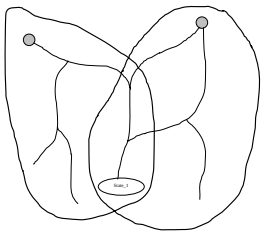

Рис. 3.18: Пример проявления эффекта компактизации на бóльших участках

Возможно, что человек продублировал бы “пересекающийся” участок кода для получения более читабельной конструкции. Автоматически это сделать тяжело, поскольку это зависит только от конкретной задачи, но важно предоставлять человеку такую возможность. Поэтому мы даем ему возможность редактировать конечный автомат до генерации SDL кода. При данном редактировании человек модифицирует событийную структуру автомата и задает необходимую структуру будущего SDL кода.

3.6.6 Выделение процедур

Опыт практического программирования показывает, что в реальных задачах достаточно часто используются процедуры. Это удобно и из-за повторяемости участков кода, и из-за соображений его компактизации и увеличения читабельности.

Достаточно очевидным кандидатом на процедуру является гамак, то есть область в графе в которой выделены две (различающихся) вершины. Одна из них называется входом, другая — выходом. Любой путь через вершины гамака проходит через вход и через выход. Схематично это изображено на рисунке 3.19. Алгоритмы по выделению подобных областей описаны в [8][9].

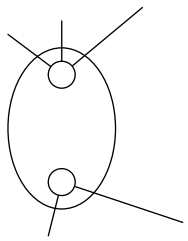

Рис. 3.19: Идея выделения процедур

Дополнительным условием на такую область является отсутствие завершения автомата внутри нее, поскольку в SDL коде в процедуре объект не может завершаться. Рассматривая автомат как граф[10], все такие области можно найти и сравнить их на эквивалентность. Две области будем считать эквивалентными, если равны языки автоматов, начальной вершиной которых является вход области, а завершающей — выход из области. Для этого сравнивается эквивалентность соответствующих минимизированных автоматов. Эквивалентные области — кандидаты на процедуру.

Замечание 1

Поясним, почему выделение процедур может дать результаты после алгоритма минимизации автоматов. Рассмотрим ситуацию на рисунке 3.20. Алгоритм минимизации с данным конечным автоматом ничего не сдела-

[9] Данные алгоритмы также использовались для оптимизации программ.

[10] Здесь для технологического средства неявно возникает задача изображения графа, которой посвящены множество работ [32].

ет. Теперь участки $S1$ можно будет заменить на конечный автомат так, что алгоритм минимизации тоже ничего сделать не сумеет, а выделение процедуры будет возможно.

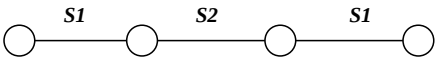

Рис. 3.20: Данный автомат не изменится при работе алгоритма минимизации

Замечание 2

Выделение процедур сильно завязано на процедуры деоптимизации автомата и должно выполняться под контролем человека. Ибо достаточно просто можно построить пример (см. рисунок 3.21), когда минимизация автомата делает невозможным выделение процедур, а небольшая деоптимизация может дать возможность выделить процедуры и, как следствие, повысить читабельность SDL кода.

Пусть в конечном автомате существуют два гамака, но из-за процедуры минимизации автомата второй гамак был “испорчен” и существует путь через его выходящую вершину, не пролегающий через начальную. Эта ситуация показана на рисунке 3.21. На нем возможно вывести дополнительный путь за границы гамака с помощью дублирования вершин и дуг пути. В этом случае автомат станет менее оптимальным, но можно будет выделить процедуры. Автор считает, что подобные вещи должен находить человек, причем идеи для их нахождения следуют как раз из логики разрабатываемой системы.

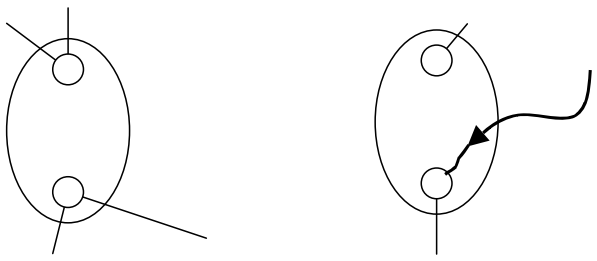

Рис. 3.21: Ручное редактирование улучшает возможности по выделению процедур

Процесс выделения процедур следует производить:

- в диалоговом режиме, поскольку пользователь должен решать, из каких областей нужно делать процедуры;
- с учетом вложенности (начинать с “маленьких” участков и переходить к объемлющим);
- следует иметь настраиваемый параметр, начиная с какого количества элементов гамак претендует на то, чтобы быть процедурой.

После выделения процедур, с точки зрения алгоритма генерации, их, как и в случае с таймерами, следует обрабатывать как замену на посылку специального сообщения, а потом замену обратно. Конечный автомат процедуры обрабатывается аналогично генерации SDL кода для всего конечного автомата, с точностью до минимальной замены элементов процесса на элементы процедуры.

3.6.7 Детерминининизация по выходящим сообщениям

Минимизация автомата может ухудшать порождаемый SDL код за счет того, что автомат детерминизируется по всем сообщениям, а следовало бы иногда его детерминизировать лишь по входящим сообщениям. Код, созданный человеком, иногда является недетерминированным по выходящим сообщениям. Соответствующий пример был рассмотрен в разделе 3.1.3 на рисунках 3.2, 3.3, 3.5, 3.6.

Подобная проблема может быть решена следующими способами:

1. замена некоторых “мешающих” выходящих сообщений на новые сообщения, не присутствующие на автомате, перед детерминизацией и минимизацией, а после них замена обратно. Таким образом часть сообщений исключается из детерминизации;
2. замена целой области на новое сообщение, проведение детерминизации и минимизации, а затем заново расшифровка введенного сообщения в область. Таким образом вся область защищается от изменений.

Следует осуществлять подобные выделения с помощью программиста в диалоговом режиме.

3.6.8 Параллелизм

Если рассматривать MSC стандарт в полном объеме, а не на множестве, выбранным нами для достаточной демонстрации разработанных алгоритмов, то окажется, что конструкции опционального исполнения, циклы, корегионы не меняют ровным счетом ничего. Как и применение HMSC вместо MSC. Для них необходимым образом меняется алгоритм извлечения конечного автомата из (H)MSC диаграммы, а далее работают ровно те же самые алгоритмы без каких-либо изменений. Только одна конструкция требует вмешательства в алгоритм — это конструкция параллельного исполнения.

На данном этапе сразу введем ряд ограничений на данную конструкцию:

- не должно быть входящих сообщений для данного объекта, которые относятся сразу хотя бы к двум параллельно исполняющимся сценариям, поскольку в этом случае не будет ясно, развитию какого сценария отвечает приход данного сообщения;

- объект не должен завершать выполнение в одном сценарии и продолжать какую-либо деятельность в других.

Данные ограничения введены для алгоритма генерации с точки зрения верификации входных данных. Хотя в некоторых исключительных случаях SDL код может быть создан вручную с нарушением данных ограничений.

Если требуется создать автомат, который будет параллельно исполнять несколько конечных автоматов, то он строится с помощью декартового произведения автоматов. Количество состояний получившегося автомата будет равно произведению числа состояний в каждом из параллельно исполняющихся автоматов. Отметим, что если автоматы более-менее содержательны, то число состояний окажется значительным для визуальной работы человека с получившейся структурой.

В случае с построением соответствующего SDL кода, нельзя перемножать автоматы до генерации, поскольку результирующий код будет содержать много условных выражений вида “стоит ли послать сообщение X согласно сценарию A, либо перейти к ожиданию сообщений сценария B, а затем лишь послать X”. Нужно перемножать уже построенный SDL код. Объясним на примере из двух параллельных сценариев. Один условно назовем “горизонтальным”, а другой — “вертикальным”. Каждый из них порождает SDL код — “горизонтальный” или “вертикальный”. По аналогичным алгоритмическим соображениям, как и для декартового произведения автоматов, мы получим произведение количества SDL состояний в каждом из SDL описаний. Данное декартово произведение представимо в виде прямоугольника из точек — матрицы. При получении сообщения для “горизонтального” сценария идет выполнение действий “горизонтального” SDL кода, что означает сдвиг по горизонтальной составляющей матрицы. Если для “вертикального”, то все вертикальное. Характеристиками данного подхода являются:

+ возможность изменять поведение по одному сценарию в зависимости от текущей ситуации в другом сценарии;

+ возможность начинать исполнение одного сценария только после достижения определенного состояния в другом сценарии;

– для содержательных SDL описаний произведение количества состояний выше человеческого восприятия.

С точки зрения автора, последний минус перевешивает все плюсы. Хотя и возможно на гигантской схеме вручную урезать количество состояний — например, последовательное выполнение двух сценариев так же является одним из предельных случаев параллельного исполнения. Поэтому данный подход остается лишь в виде схемы и дополнительно не детализируется.

Введем еще одно ограничение. Пусть параллельные сценарии исполняются без знания о состоянии других сценариев. При этом они запускаются сразу при входе в соответствующую область параллелизма и выходят из нее только тогда, когда все сценарии будут выполнены.

Пользуясь всеми тремя ограничениями, можем достаточно просто реализовать исполнение параллельных сценариев. Для простоты пусть у нас есть два параллельных сценария, выполняющихся в некоторой области функциональности. Тогда создаем три участка SDL кода, которые соответствуют:

1. исполнению 1-го параллельного сценария;
2. исполнению 2-го параллельного сценария;
3. процедуре, которая:
    - при входе в нее запускает SDL код каждого из параллельных сценариев в виде отдельных объектов либо сервисов, что зависит от деталей реализации;
    - входящие сообщения, относящиеся к 1-му либо 2-му сценарию пересылает на соответствующую сущность;
    - завершается тогда, когда завершатся сущности, соответствующие обоим сценариям.

Для большего количества сценариев данная схема расширяется элементарно.

3.6.9 Обобщенный алгоритм разработки динамики системы

Теперь подведем итоги. У базового алгоритма синтеза, описанного в разделе 3.3, есть ряд недостатков. Наиболее существенными являются:

1. минимизация автомата порождает склеивания участков SDL кода там, где человек бы их не сделал;
2. отсутствие разумной компактизация SDL кода в виде выделения процедур;
3. требование к детерминизации автомата по выходящим сообщениям ухудшает код;

4. отсутствие возможности изменять генерируемый SDL код для более удобного его восприятия.

Данные проблемы решаются с помощью введения промежуточной модели между MSC и SDL. Это модель конечных автоматов, которую разрешаем редактировать до и после связки процедур детерминизация–минимизация. Соответствующие конечные автоматы можно "переразложить" тем или иным способом перед процедурой генерации SDL для повышения читабельности кода. При этом динамика системы создается следующим образом:

1. на MSC диаграммах создаются алгоритмы взаимодействия различных частей системы друг с другом и с внешним миром;

2. отдельно рассматривается динамика каждого из объектов. Вся картина мира разбивается на поведение данного объекта и весь остальной мир. На данном этапе рассматривается только событийно-ориентированный механизм, причем его удобно представлять в виде конечного автомата — графа [32]. Частям данного конечного автомата можно придать требуемый вид для того, чтобы в конечном SDL коде алгоритм был записан в том или ином виде. На данном этапе существуют только последовательности сообщений с упорядочивающей их логикой. Разбиение на состояния SDL и дополнительный код еще не проводилось;

3. после преобразования конечного автомата в нужный вид по нему создается SDL код с помощью алгоритма генерации. На данном этапе дополнительно к сообщениям уже есть такие стандартные атрибуты SDL как состояния, операторы ветвления и процедуры. На данном этапе еще нет конкретных переменных;

4. заключительный этап разработки. На нем SDL конструкции наращиваются работой с переменными и конкретными операторами.

Именно подобный подход, с точки зрения нашей системы, отражает постепенность процесса создания динамики системы.

3.7 Выводы

В данной части представлен новый метод, который обеспечивает все возможности по автоматическому синтезу и при этом позволяет управлять данным процессом для настройки SDL кода для использования человеком. Задача по настройке кода впервые была поставлена автором.

Приведенные алгоритмы по выделению процедур и обработке параллельных сценариев не встречались в существующих работах. Однако в реальных работах MSC диаграммы программируются с учетом параллелизма, а SDL диаграммы создаются с использованием процедур. Поэтому мы должны обеспечить их поддержку в алгоритмах синтеза. Выделение процедур позволяет улучшить SDL код и облегчить его сопровождаемость. Автоматическая генерация SDL кода по параллельным сценариям позволяет очень просто избежать этого достаточно "болезненного" в традиционных методах для программиста места.

Хотя основным предназначением предложенного комплекса является переход от MSC к SDL, но он с таким же успехом применяется для оптимизации уже написанного SDL кода. Так же он будет обеспечивать переход к SDL от любой модели, сводящейся к конечно–автоматной. Помимо этого он применим в ряде других технологий, где SDL модель является промежуточным этапом [45, 37, 61, 52]. Использование предложенного метода в этих работах позволит проектировать программно-аппаратные комплексы на языке MSC с описанием взаимодействий ПО, аппаратуры, и, что особенно ценно, стыка между ними.

Разработанный метод повышает производительность разработчика систем в реальных работах, в особенности — при переходе от стадии проектирования к программированию.

Глава 4

Модификация MSC диаграмм для описания обратных веток

В разработанном технологическом средстве REAL используются и MSC, и SDL диаграммы, реализован синтез SDL диаграмм по MSC. С помощью данного средства был создан ряд телефонных станций. Накопленный опыт показал, что у существующих MSC диаграмм есть ряд недостатков, которые не позволяют создать исчерпывающее описание поведения объекта со всеми обратными ветками. Такое описание было бы очень полезно:

- как исчерпывающая документация, вместо набора разрозненных веток;
- как модель, по которой можно проводить автоматическую проверку соответствия SDL реализации MSC документации;
- как предварительная заготовка для автоматической генерации SDL диаграмм.

Для начала продемонстрируем, с какими типовыми сложностями сталкивается проектировщик при разработке MSC диаграмм, а затем продемонстрируем расширение MSC диаграмм, с применением которого удобнее разрабатывать системы. Данное расширение обладает всеми возможностями

существующих MSC диаграмм, а также обладает дополнительной гибкостью, что позволяет строить исчерпывающие MSC описания.

4.1 Пример использования текущего стандарта MSC и его обсуждение

Для начала рассмотрим процесс проектирования для небольшого примера и покажем, какие возникают трудности при использовании существующего стандарта MSC. Поставим себя на место технолога, который разрабатывает обмен сообщениями между несколькими объектами путем постепенной детализации сценария обмена.

Пусть для его задачи уже была проведена функциональная декомпозиция, и теперь проводится детализация функций на MSC диаграммах. Сначала, естественно, проектируется основное поведение системы. Для простоты будем изображать только один из объектов. Начнем его детализацию, как это изображено на рисунке 4.1.

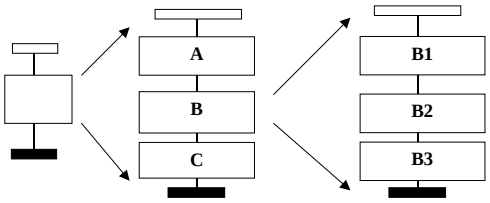

Рис. 4.1: Функциональная декомпозиция объекта

Для упрощения рисунка 4.1 стрелочками показана операция детализации функциональных блоков, что соответствует использованию конструкций ссылки и последовательного выполнения MSC диаграмм.

После выполнения этой работы начинаем проектировать обратные ветки. Под обратными ветками подразумеваем ситуации, когда поведение системы отклоняется от “идеального”. Чаще всего это ошибочные ситуации. При этом понимаем, что вместо сценария B2 могут выполниться сценарии D1 и D2. После этого оказывается, что для каждого из подсценариев сценария B завершение тоже должно быть свое - E1 и E2. Это изображено на рисунке 4.2. Таким образом, вынуждены перерисовать последнюю

детализацию с рисунка 4.1 и заменить ее на конструкцию с рисунка 4.3.

Дальше интереснее. Часто оказывается, что подобные изменения не заканчиваются текущим сценарием, а также влияют и на детализируемый сценарий. Это изображено на рисунке 4.4, где большими стрелками показаны необходимые завершения для каждого из вариантов выполнения предыдущего сценария.

Напомним, что существующие средства описания MSC не позволяют передать какую-либо информацию о выполнившемся сценарии в вызвавший сценарий. Поэтому надо переделать предыдущий (детализируемый) сценарий. А от детализирующего сценария придется отказаться. В результате получим ситуацию, изображенную на рисунке 4.5.

Очевидно, что ситуация все ухудшается и ухудшается, а сложность сценария нарастает. При этом для отображения обратных веток на текущем сценарии пришлось отказаться от преимуществ, которые давала декомпозиция и переносить все варианты на предыдущие уровни. Следует отметить, что даже перерисовка блоков, помеченных цифрами, требует усилий. В реальности надо перерисовывать десятки сообщений, из-за чего в традиционном MSC отказываются от проработок обратных веток. И даже в стандартах ITU-T часто приводят лишь прямые ветви, которые легко изобразить с помощью декомпозиции.

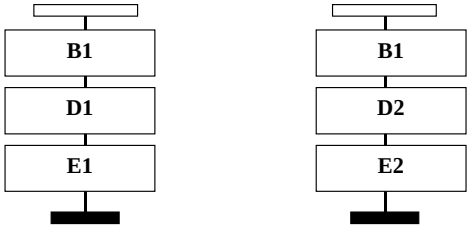

Рис. 4.2: Варианты обратных веток для блока B

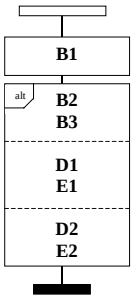

Рис. 4.3: Все варианты исполнения блока B

На этом месте позволим себе завершить пример, отметив, что в реальности все могло оказаться еще хуже по следующим причинам:

- зависимость обратных веток часто завязана еще глубже;
- при использовании нескольких объектов обработка обратной ветви для каждого из объектов могла оказаться своей.

Все это привело бы к тому, что подобные изменения включили бы в себя еще более большое количество сценариев, а также:

1. результирующая диаграмма получилась бы просто невероятных размеров, вплоть до того, что ее было бы трудно воспринимать;
2. трудоемкость ее создания и поддержания была бы очень большой.

Читатель давно должен был заметить, что к данному примеру очень хочется подойти с точки зрения языков программирования. Считать “детализируемый” блок функцией, которая возвращает значение. В “детализи-

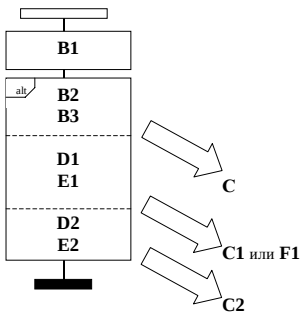

Рис. 4.4: Влияние выполнения сценария B на детализируемый сценарий с рисунка 4.1

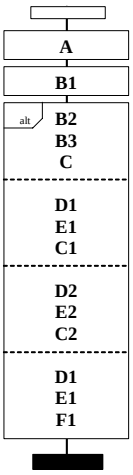

Рис. 4.5: Все варианты выполнения сценария

руемом" блоке это значение проверять и необходимым образом корректировать поведение. С точки зрения этого подхода существующий стандарт MCS реализует блок с одной точкой входа и одной точкой выхода. А хочется иметь конструкцию, которая позволяет из блока вернуть несколько результатов. В качестве парной к ней — конструкцию, позволяющую разобрать результат.

Теперь можем сформулировать основную идею предложенного метода. Мы используем модифицированные MSC диаграммы, которые могут "возвращать значения", и специальные средства анализа, которые возвращенному значению могут сопоставить некоторый набор действий. С точки зрения программиста он получает язык, который по описательным возможностям очень похож на высокоуровневые языки.

4.2 Предложенное решение

Поэтапно покажем как строится предложенное расширение MSC. Сначала опишем основные идеи. Потом покажем неформальные основы используемого математического аппарата, затем покажем как строятся блоки в процессе разбора, определим требования по корректности, и, суммируя все это, опишем весь алгоритм.

4.2.1 Краткое описание расширения

Был разработан Pascal'е-подобный язык, который стоит над MSC и позволяет создавать текстовые описания. Он предназначен для описания логики стыковки сценариев. Для начала приведем пример, а потом будем давать более подробные комментарии.

```
if f_сценарий_1=1 then
    p_сценарий_1
else
    p_сценарий_2
fi;

while f_сценарий_2>4 do begin
    p_сценарий_3;
    p_сценарий_4;
    p_сценарий_1;
end;
```

Таким образом, на данном языке обратные ветки и группы сценариев описываются более, чем просто. При этом подобное описание системы обладает ясностью, которая была просто не достижима в традиционном MSC.

Условно следует выделить два типа описаний для построения:

1. "Процедуры". Это описания, аналогичные стандартным MSC диаграммам, которые не возвращают никаких результатов. С одной стороны — это самые простые элементы, на которых строятся диаграммы что-либо возвращающие, а с другой стороны — это и то, чем в конце концов завершается объемлющее построение.

2. "Функции". Это специальным образом модифицированные описания, которые возвращают результат после своего выполнения.

Для описания оператора возвращения используем слово **return**. В примерах мы используем возвращение числа в качестве результата.

Для построения определяем следующие конструкции[1]:

- **if then else fi**
- **case**
- **for** – определенный
- **for** – неопределенный
- **while**
- **alt**
- **opt**

На значение, используемое в операторах проверки, накладываются следующие условия:

- оно должно быть известно на момент “компиляции”;
- при проверке используется результат выполнения только одного сценария. Зависимость от большего количества сценариев реализуется самим пользователем.

4.2.2 Неформальное объяснение понятий “блок” и “суперблок”

До формальных описаний по переходу от расширенного MSC описания к конечному автомату через математический аппарат блоков дадим неформальные пояснения для лучшего восприятия последующих разделов. Изображение блока приведено на рисунке 4.6.

[1]В данной работе для простоты построения считается, что все конструкции имеют Pascal’е подобный синтаксис. Построение C-подобных конструкций **switch(break); for(continue,break); while(continue,break)** также возможно. При необходимости строится и конструкция **goto**.
По аналогии с текстовыми MSC описаниями добавлены конструкции **alt** и **opt**.

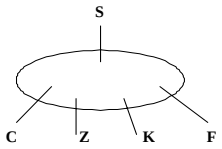

Рис. 4.6: Изображение блока

Прежде всего, блок отличается от конечного автомата тем, что состояния S и F отделены от обычных состояний. Этот шаг приближает нас к логике кода, в соответствии с которой если выполнение началось, то в начальное состояние возврат невозможен. Если выполнение завершилось, то выход из этого состояния невозможен, а не допустимые "промежуточные варианты", согласно определениям конечных автоматов, когда возможны переходы в начальное состояние и из завершающего состояния. Будем использовать эти свойства блока при проведении операций по "объединению" автоматов.

Если обычный конечный автомат имеет три типа состояний:

- начальное,
- "промежуточное",
- завершающее,

то при различных операциях агрегирования автоматов мы не работаем с уже готовым автоматом, а работаем с некоторыми неполными структурами, из которых его собираем. Соответственно, появляются структуры, которые отвечают за различные варианты агрегирования автоматов:

- K — служит для последовательного соединения блоков друг с другом, как это изображено на рисунке 4.7;
- Z — соответствует (экстренному) выходу из процедуры;
- C и R дают возможность выходить из функции с сообщением результата выхода.

Уточним, что блок соответствует участку кода. При разборе различных участков кода последовательно агрегируем блоки необходимым образом. Например, на рисунке 4.7 показана конкатенация блоков по K, а на

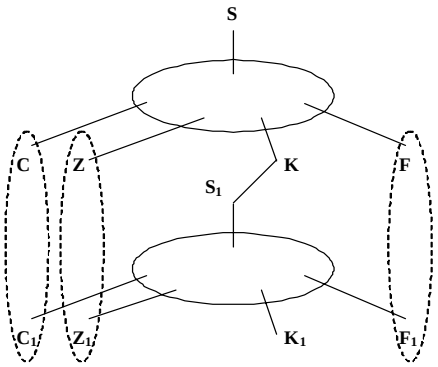

Рис. 4.7: Конкатенация блоков по K

рисунке 4.8 показан пример обработки оператора **alt** с помощью операции конкатенации по A. И в том, и в другом построении снова получаем блок, поскольку множества C, Z, R, F объединяются. Схематично это показано на рисунке 4.7.

Отдельно отметим причину наложения условий по отсутствию пересечений в множествах вида $(Q \setminus (\{S, K\} \cup C \cup Z \cup F))$. На самом деле это условие обозначает то, что мы агрегируем разные блоки, а не, например, блок с частью его самого. Графически это условие совершенно очевидно и не нуждается в проверке. Исключения из него бывают лишь при использовании Condition's, которые позволяют переходить из одного места в другое.

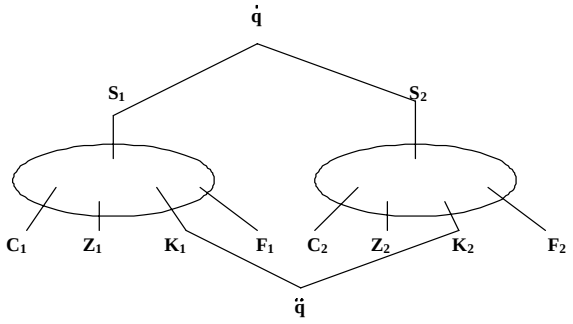

Рис. 4.8: Пример построения операции alt

Также в процессе построения возникает понятие суперблока. Как было отмечено в определении, у суперблока несколько выходов вида $K^1, \ldots, K^n$. Обычный блок является суперблоком уровня 1 и имеет один выход. У суперблока уровня n таких выходов n.

В наших построениях суперблок возникает при разборах функций, пример этого изображен на рисунке 4.9. Суперблок является заготовкой для

построения блока. На рисунке 4.9 на основе суперблока пунктирными линиями строится блок при разборе конструкции **if**. Также отметим разбиение множества C на два подмножества согласно выражению в нетерминале **if**.

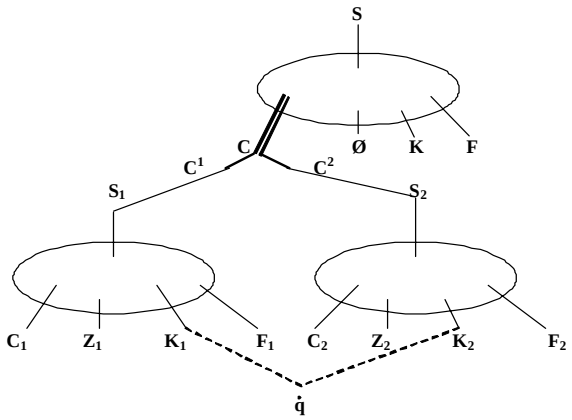

Рис. 4.9: Построение суперблока и переход от него к блоку при разборе конструкции if

4.2.3 Построение блоков при разборе грамматики

Пусть на входе синтаксического анализатора [1] имеется текст, содержащий расширенное MSC описание. На данный текст накладывается требование корректности, которое будет более детально объяснено в разделе 4.2.5. Покажем, как по такому описанию создается конечный автомат. Для этого, согласно грамматике, приведенной в Приложении В, покажем, как обрабатываются основные моменты разбора.

Основная наша идея — это понятие блока. Мы показываем, как из различных операторов строится блок, как цепочка операторов также дает блок, что MSC диаграмма согласно стандарту также является блоком, что описание процедуры можно превратить в простой блок. Поэтому результирующая процедура либо в виде обычного описания MSC, либо в виде расширенного, является простым блоком, а из него мы уже строим конечный автомат.

Не вдаваясь детально в теорию множеств, считаем, что при построении нового выражения (statement) всегда можем взять состояния, которые на текущий момент еще не участвовали ни в каких построениях, если спе-

циально не оговорено обратное — использование Condition. Это дает возможность считать, что необходимое условие на использование операций конкатенации — пустота пересечения множеств — выполнено.

Пустое выражение

По определению мы считаем его блоком: $B_{empty} = (\{\dot{q}, \ddot{q}\}, \emptyset, \emptyset, \{(\dot{q}, \ddot{q})\}, \dot{q}, \emptyset, \emptyset, \emptyset, \ddot{q}, \emptyset)$.

Выражение return

Вводим блок: $(\{\dot{q}, \ddot{q}, \dddot{q}, \ddddot{q}\}, \emptyset, \emptyset, \{(\dot{q}, \ddot{q}), (\ddot{q}, \dddot{q})\}, \dot{q}, \emptyset, \emptyset, \{\ddddot{q}\}, \ddddot{q}, \emptyset)$.[2]

Выражение return number

Вводим блок:
$(\{\dot{q}, \ddot{q}, \dddot{q}, \ddddot{q}\}, \emptyset, \emptyset, \{(\dot{q}, \ddot{q}), (\ddot{q}, \dddot{q})\}, \dot{q}, \emptyset, \{\ddddot{q}\}, \emptyset, \ddddot{q}, \{(\ddddot{q}, number)\})$.

Выражение if

1. Поскольку выражение if . . . then . . . fi сводится к if . . . then . . . else ; fi, то с самого начала полагаем, что у if есть else часть.

2. При обработке нетерминала expression имеем:

   - функцию, заданную в нем (B);
   - оператор сравнения;
   - число.

   Обработка тела оператора if дает два блока. Один из них соответствует then части (B_1), а другой — else (B_2). Применяем оператор сравнения (expr) к отношению R функции B. Это дает разбиение множества C на два непересекающихся подмножества:
   $C^1 = \{q : expr(n) = true, (q, n) \in R\}$,
   $C^2 = \{q : expr(n) = false, (q, n) \in R\}$.

3. Строим суперблок порядка 2: $\tilde{L} = \bigotimes_X(B, B_1, B_2)$

4. По суперблоку $\tilde{L} = (\tilde{Q}, \tilde{V}, \tilde{T}, \tilde{E}, \tilde{S}, \tilde{F}, \tilde{C}, \tilde{Z}, \tilde{K^1}, \tilde{K^2}, \tilde{R})$ и $\dot{q} \notin \tilde{Q}$ строим блок $(\tilde{Q} \bigcup \{\dot{q}\}, \tilde{V}, \tilde{T}, \tilde{E} \bigcup \{(\tilde{K^1}, \dot{q}), (\tilde{K^2}, \dot{q})\}, \tilde{S}, \tilde{F}, \tilde{C}, \tilde{Z}, \dot{q}, \tilde{R})$.

[2]Следует понимать, что в каждом из построений выражения символы $\dot{q}$ и аналогичные являются разными, и отношения друг к другу не имеют согласно вышесделанному замечанию.

Выражение case

1. Поскольку case корректное, то мы имеем функцию и имеем разбиение ее множества вершин, маркированных как выход с номером, на непересекающиеся подмножества в количестве n штук. Каждый из вариантов case, возможно, с веткой else, дает нам блок.

2. Аналогично случаю с if, строим суперблок порядка n $L = (Q, V, T, E, S, F, C, Z, K^1, \ldots, K^n, R)$, используя функцию, разбиение и n блоков.

3. По суперблоку и $\dot{q} \notin Q$ строим блок $(Q \bigcup \{\dot{q}\}, V, T, E \bigcup \{(K^1, \dot{q}), \ldots, (K^n, \dot{q})\}, S, F, C, Z, \dot{q}, R)$.

Выражение while

1. На входе имеем функцию, разбиение и два блока кода.

2. Строим по ним суперблок $L = (Q, V, T, E, S, F, C, Z, K^1, K^2, R)$.

3. По суперблоку и $\dot{q} \notin Q$ строим блок $(Q \bigcup \{\dot{q}\}, V, T, E \bigcup \{(\dot{q}, S), (K^1, S)\}, \dot{q}, F, C, Z, K^2, R)$.

Выражение for–бесконечный

При разборе тела данного оператора получаем блок $(Q, V, T, E, S, F, C, Z, K, R)$. Пусть $\{\dot{q}, \ddot{q}\} \bigcap Q = \emptyset$, тогда блоком, реализующим оператор for–бесконечный, мы назовем блок $(Q \bigcup \{\dot{q}, \ddot{q}\}, V, T, E \bigcup \{(\dot{q}, S), (K, S)\}, \dot{q}, F, C, Z, \ddot{q}, R)$.

Выражение for–конечный

Поскольку данный оператор представляет собой несколько раз повтор блока, то необходимый блок повторяем соответствующее количество раз.

Выражение alt

При разборе данного нетерминала получаем набор блоков $B_1, \ldots, B_n$. Тогда блоком, реализующим данный оператор, обозначим блок $B = \bigotimes_A (B_1, \ldots, B_n)$.

Выражение opt

При разборе тела данного оператора получаем блок B_1. Тогда блоком, реализующим данный оператор, обозначим блок $B = \bigotimes_A(B_1, B_{empty})$.

Нетерминал procedure_declaration

Согласно замечанию к Определению 16 по процедуре строим простой блок.

MSC диаграмма согласно стандарту

Поскольку из данной диаграммы можно получить конечный автомат, то она представляет собой простой блок. Данный пункт вкупе с предыдущим позволяют не различать MSC диаграммы согласно стандарту и описания процедур в расширенном описании MSC.

Вызов процедуры

Поскольку процедура дает простой блок, то при использовании выражения мы этот блок и используем.

Добавление statement к list_statement

Уже было показано, что любое выражение является блоком. Соответственно, когда list_statement представляет из себя только одно выражение, то он уже является блоком. Когда к нему добавляется еще один блок, то используем операцию конкатенации по K, в которой блок, равный list_statement, используем в качестве первого операнда, а присоединяемое выражение является вторым операндом. После этого он снова является блоком.

4.2.4 Совместное использование графического и текстового описаний

Разумеется, хочется иметь возможность совместного использования графического и текстового описаний. Существует три способа их состыковки.

1. На графических описаниях можем использовать описание процедуры в виде текстового расширения так же, как и reference/inline конструкции.

2. В текстовых описаниях используем MSC описания так же, как и текстовые процедуры, поскольку и те, и другие порождают простой блок.

3. На графические описания может добавляться символ return [number]. Графически он изображается в виде стрелки, возможно, с кодом возврата. Соответствующие MSC диаграммы будем именовать обобщенными. Данный пункт является опциональным, поскольку расширяет “чистое” графическое описание. Поэтому этот пункт дан только на уровне идеи (см. рисунок 4.10). Подобная конструкция эквивалентна текстовому описанию функции.

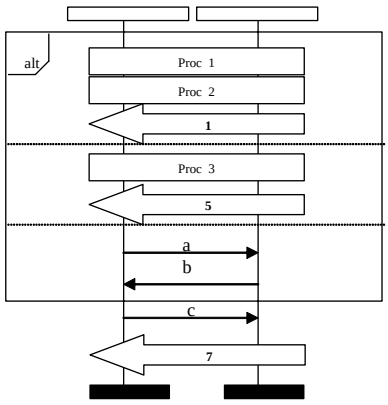

Рис. 4.10: Расширение графических диаграмм для добавления возможности возвращать результат

4.2.5 Построение конечного автомата по расширенному MSC описанию для выделенного объекта

Прежде чем начинать какие-либо построения, надо убедиться, что они имеют смысл. Другими словами, должны провести ряд проверок на корректность.

1. На входе должны быть синтаксически правильные конструкции.

2. Опционально проводится проверка отсутствия ошибок в самой модели. Соответствующие алгоритмы были разработаны другими авторами, поэтому просто на них сошлемся. Чуть более детально это освещено в разделе 4.4.2.

3. Проверяется корректность конструкций и их использования:

- все блоки являются корректными;
- нетерминалы procedure_declaration получаются из блоков процедуры;
- нетерминалы function_declaration получаются из блоков функций;
- в case используется разбиение множества значений, возвращаемых функцией, на непересекающиеся подмножества, дающие в объединении все множество;
- выдаются предупреждения, если for–бесконечный порождает блок, у которого $C \bigcup Z = \emptyset$.

Для построения конечного автомата нужна дополнительная информация — мы должны знать, с какой процедуры должны начинать описание данного объекта. Процедурой может являться как обычная MSC диаграмма, так и процедура из расширенного MSC описания. Описание объекта с функции начинаться не может.

Для объекта можно выделить, как его обычные диаграммы, процедуры и функции связаны друг с другом с помощью:

- reference и inline конструкций на обычных MSC диаграммах;
- использования MSC диаграмм из текстового описания;
- вызовов текстовых процедур из текстового описания;
- вызовов текстовых процедур из MSC диаграмм;
- использование текстовых функций из текстового описания MSC;
- использования обобщенных MSC диаграмм в качестве функций.

Многократное использование, например, из одной процедуры другой процедуры, считается как несколько связей. Вся совокупность зависимостей объекта образует несвязный ациклический направленный граф [9]. При построении конечного автомата надо указать, с какой процедуры его надо начать строить. Автоматически определить, какая процедура является

“начальной”, невозможно, поэтому она задается. После этого на множестве зависимостей организуется порядок и мы получаем ациклический граф, начинающийся с выбранной процедуры. Если в нем присутствуют циклы, то это значит, что присутствует рекурсия и описание не корректно. Порождение автомата невозможно. Если оказались, что некоторые диаграммы оказались недостижимыми, надо выдать соответствующее предупреждение.

Данный ациклический граф зависимостей подвергается копированию процедур и функций начиная с листьев до того момента, чтобы на любую процедуру или функцию имелась ссылка не более, чем с одного места. Данную операцию будем именовать расклейкой графа зависимостей. В результате получаем дерево зависимостей.

После этого строим простой блок для выбранной процедуры. Данный блок будет включать в себя другие блоки процедур и функций согласно дереву зависимостей. По данному простому блоку строим конечный автомат. Данный конечный автомат так же можно проверить на корректность:

- все ли состояния достижимы из начального состояния. Если существуют недостижимые, следует выдать предупреждение;
- есть ли хоть одно завершающее состояние, достижимое из начального.

Проведем весь алгоритм, заданный по шагам. На вход он получает выбранный объект, набор MSC диаграмм и расширенное описание MSC. В результате его выполнения получаем конечный автомат. Шаги выполнения:

1. проверить корректность синтаксиса;
2. проверить корректность конструкций и их использования;
3. выполнить расклейку графа зависимостей;
4. последовательно вычислить все блоки;
5. по блоку выбранной процедуры построить конечный автомат;
6. проанализировать конечный автомат на корректность.

4.3 Примеры

Приведем несколько примеров, иллюстрирующих предложенный подход.

4.3.1 Построение блоков по коду

Рассмотрим несложный пример, показывающий построение блоков. Из-за того, что согласно алгоритму, это динамический процесс, на приведенном примере некоторые детали могут отсутствовать. Ниже приведена простая функция, а на рисунке 4.11 показана схема построения ее блока.

```
function test;
begin
while f1=2 do
        if f2>0 then
            return 1
        else
            p1
        fi;
return 2;
end
```

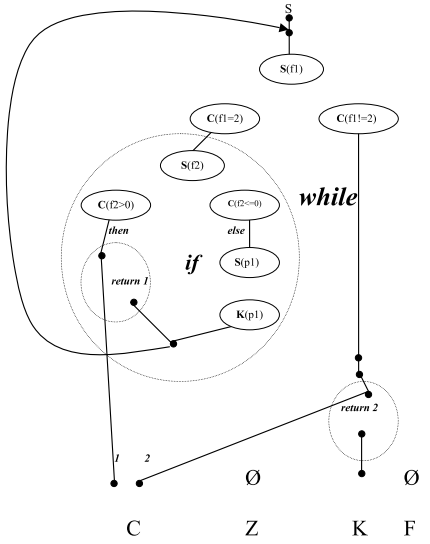

Рис. 4.11: Схема построения блока по функции test

4.3.2 Описание примера из раздела 4.1 с помощью предложенного расширения MSC

Пусть все необходимые сценарии уже описаны в виде MSC диаграмм. Тогда продемонстрируем запись детализируемых сценариев и главной процедуры через предложенное расширение.

```
function B_middle;
begin
alt_begin
    begin
        B2;
        return 0;
    end
alt
    begin
        D1;
        return 1;
    end
alt
    begin
        D2;
        return 2;
    end
alt_end
end


function B;
begin
B1;
case B_middle of
    0 : begin
            B3;
            return 0;
        end;
    1 : begin
            E1;
            return 1;
        end;
```

```
    2 : begin
            E2;
            return 2;
        end
esac
end

procedure BC;
begin
case B of
    0 : C;
    1 : alt_begin
            C1
        alt
            F1
        alt_end;
    2 : C2
esac
end

procedure main;
begin
    A;
    BC;
end
```

Достаточно очевидно, что подобное описание более ясно отражает суть происходящего и гораздо более удобно модифицируется.

4.4 Замечания к использованию

Обсудим предложенную модель с точки зрения ее применимости в разных практических ситуациях и прокомментируем полученный результат.

4.4.1 Использование Condition

По мнению автора, Condition является очень неудобной конструкцией. При появлении Condition с одним и тем же именем в диаграммах больше двух раз поведение спецификации становится крайне запутанным. Автор рекомендует вообще воздержаться от использования данной конструкции и пользоваться предоставляемыми высокоуровневыми конструкциями **if, for, while**.

При использовании Condition пользователем считается, что он понимает, что он получит в результате взаимодействия его конструкции и высокоуровневых конструкций. Например, при записи ***for*** *4* ***do*** *сценарий_N;* будет проведена развертка цикла в 4 последовательных выполнения сценария. Если в сценарии_N будет использоваться Condition — это можно будет трактовать и как возможный переход из 1-го выполнения сразу в 4-й (а также и обратно, и куча других побочных эффектов). Теоретически все подобные проблемы можно выявить и специальным образом переименовывать все Condition, чтобы не возникало побочных эффектов, но результат минимальный. Проще запретить совсем. Это как в АЯВУ — вы можете дополнительно пользоваться ассемблером, регистрами, стеком. Это подразумевает достаточный уровень понимания результата.

4.4.2 Корректное проектирование

Рассмотрим пример в стандартном виде MSC диаграмм — рисунок 4.12. На нем изображены два варианта обмена:

1. если объект 3 отвечает сообщением C, то первый объект посылает сообщение E;

2. если отвечает D, то посылается сообщение F.

Чудес не бывает. Поэтому первый объект каким-то образом должен решить, какое сообщение он должен отослать. Неявным образом информация не передается. И откуда-то первый объект должен узнать о том, какой из обменов сообщениями произошел между объектами 2 и 3. У него может

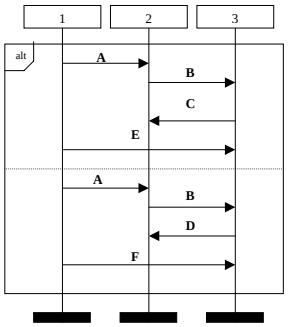

Рис. 4.12: Ситуация с нелокальным выбором

быть данная информация — например, она могла быть передана в предыдущих сообщениях, а могла и не быть.

В том случае, когда используется высокоуровневый метод проектирования, все абсолютно аналогично. Поэтому достаточно наивно верить, что можно проанализировать конструкцией **if** обмен, произошедший между двумя объектами, а затем попытаться изменить поведение какого-то третьего объекта без явной передачи ему информации о случившемся.

```
if объекты_K_L_поступили=1 then
    объекты_M_N_сценарий_1
else
    объекты_M_N_сценарий_2
fi;
```

В этом случае в сгенерированной SDL модели поведения нужно будет заполнить выбор решения, а информация для принятия данного решения будет отсутствовать. В англоязычной литературе такая проблема носит название “non-local [branching] choice”, и алгоритмы нахождения таких проблемных мест опубликованы в [30, 34]. Соответственно, можно анализировать построенные MSC спецификации на обнаружение подобных мест и выдавать предупреждающие сообщения.

Отметим, что даже если исходная MSC модель была корректной, то получившаяся SDL модель может содержать проблеммные места из-за перехода от модели совокупного поведения каждого из объектов к моделям поведения каждого из объектов. Детальнее это рассмотрено в работе [59].

4.5 Сравнение с существующими работами

Среди различных диаграмм, с которыми можно было производить сравнение, были выбраны диаграммы, позволяющие:

1. отобразить алгоритмы взаимодействия нескольких объектов.

   Из-за этого некоторые модели, типа SDL [48] и Statechart диаграмм [64] не рассматривались, поскольку они отображают механизм внутреннего принятия решений объектом, а не алгоритмы взаимодействия.

2. в идеале позволяют получить описание всего поведения объекта, а не только некоторого информационного среза, как например Use Case [64] диаграммы.

Рассмотрим следующие диаграммы, при этом они объединены в группы, поскольку многие из них схожи с точки зрения применимости для решения описанной проблемы.

1. MSC подобная группа:

   - MSC [49]/HMSC [46];
   - UML Sequence [64] и UML Collaboration [64] диаграммы.

2. Chisel Diagrams [31] — попытка представить поведение всей системы в качестве набора состояний конечного автомата. В состояниях записываются выполняемые действия, а переходы соответствуют событиям - условиям. Общая модель — вариант конечного автомата, где действия производятся в состояниях.

3. Somé's Scenarios [71]. Следует рассматривать как вариант MSC с добавлением времени и пред- и постусловий.

4. Потоки данных и их связь со структурами:

   - UML Activity диаграммы [64];

- Use Case Maps (UCMs) [29].

Описываются потоки данных, их взаимодействие и возможные варианты поведения.

5. Life Sequence Charts (LSCs) [36]. Модель, построенная на основе MSC, в нее вкладывались возможности по контролю за возможными ошибочными ситуациями, что и вызвало расширение стандарта MSC.

Все перечисленные подходы позволяют провести декомпозицию "куска поведения" на более мелкие части. Здесь прослеживается полная аналогия с разработкой статической модели "сверху-вниз". В ней сначала рассматривается вся система, затем она делится на более мелкие части. Данный процесс прекращается на блоках, которые уже не надо детализировать.

Динамика системы является более сложной частью, и опыт автора в сфере телекоммуникаций показывает, что очень сложно разбить систему на шаги так, чтобы можно было последовательно выполнять расписанные шаги, и нигде не требовалось модифицировать свое поведение в зависимости от результата на предыдущем шаге. Скорее прослеживается даже обратная зависимость — всегда приходится закладываться на то, что одним из результатов выполнения блока может быть ошибка и, в зависимости от ее серьезности, надо изменять свое поведение. Из всех вышеперечисленных моделей описания поведения системы только одна предусматривает возможность такого описания. Это LSCs — попытка улучшить MSC диаграммы для реального применения. Она развивалась также для более гибкого описания реальных сценариев. И в нем проводится попытка выделять классы по корректности поведения. Одно из средств выделения — Condition, которые делятся на обязательные (hot) и желаемые (cold). Нарушение обязательного Condition приводит к завершению по ошибке, а нарушение желаемого — выход из текущего сценария.

В результате в данной модели сценарий уже имеет два варианта завершения — нормальный и ошибочный[3]. Предлагаемая модель позволяет

[3]Это красивая идея с полным завершением системы по обнаруженной ошибке, но в реальных системах это неприменимо, и ставится цель сохранить часть системы и перезапустить остальное. Поэтому ситуацию с "глобальной ошибкой" мы просто не рассматриваем.

гораздо более гибко рассматривать возможные варианты выполнения сценария и поэтому является более удачным подходом к описанию полного поведения системы.

Моделью, лежащей несколько в стороне, является развивающаяся модель CTP (Communicating Transaction Processes), теоретическое описание которой дано в [63], а описание средств программной поддержки в [2]. Идейно данная модель состоит из объединения сетей Петри с MSC диаграммами. В данной модели сети Петри описывают поведение каждого процесса, дающих в объединении поведение всей системы. Состояние, описывающее взаимодействие нескольких процессов, называется транзакционной схемой и детализируется в виде MSC диаграмм. Таким образом, описание системы является двухуровневым — на верхнем уровне описывается поведение системы в виде сетей Петри для процессов, а уровнем ниже описывается их детализация на MSC диаграммах. Общим данной модели и модели, предложенной в данной работе, является то, что в качестве базовой структуры используется MSC, а дальше создается надстройка. В модели CTP надстройка создается с помощью сетей Петри, а предложенной — используются структуры АЯВУ. С точки зрения автора, выигрыш дает разница между моделями АЯВУ и сетей Петри из-за того, что технологам при разработке протоколов привычно думать в терминах циклов и условных операторов, нежели в терминах сетей Петри. Так же в модели CTP отсутствует возможность поменять поведение процесса, описанное в виде сети Петри, в зависимости от результата выполнения сценария, что необходимо при создании реальных задач.

Для полноты картины следует упомянуть о существовании текстовых языков, нацеленных на описание взаимодействия различных объектов. Достаточно полный обзор (Pascal–FC, PARSEC, Occam, Ada, Java, Estelle, Lotos, UNITY, Esterel, HardwareC, Handel–C) можно найти в работе [28]. Соответствующие решения не решают задач рассматриваемой проблемы, поскольку они предназначены для описания поведения одного объекта. Это можно прокомментировать тем, что данные языки “выросли” из языков программирования и нацелены на реализацию — в идеальном варианте на создание описания, которое можно откомпилировать и исполнить.

Расширение MSC диаграмм, предложенное в данной работе, структурирует описательные возможности MSC диаграмм путем надстраивания над ними еще одного уровня, оставаясь ступенью выше над реализацией.

4.6 Выводы

В завершение данной главы кратко перечислим все плюсы предлагаемого подхода:

- дается возможность более гибко описывать обратные ветви;
- он позволяет более понятно и компактно описывать взаимодействие сценариев, используя механизм сообщений из MSC и конструкции из операторов, подобные конструкциям из АЯВУ;
- после получения исчерпывающих MSC описаний в рамках технологии можно получить заготовки SDL кода, доработка которых для реального SDL минимальна;
- внедрение подобного описания как стандарта позволит более гибко использовать MSC диаграммы и создавать на базе них исчерпывающие стандарты для телекоммуникационных систем.

Данный подход значительно усиливает возможности генерации и верификации, предложенные в данной работе, и повышает возможности стадии проектирования технологии REAL.

Заключение

В диссертации предложена модель стыковки MSC и SDL диаграмм на базе технологии REAL. Последовательно рассматриваются три проблемные области:

- ▷ верификация SDL программ по MSC документации;
- ▷ генерация SDL диаграмм по MSC диаграммам;
- ▷ получение полного описания поведения объекта на MSC модели.

В данных областях были предложены следующие решения:

- разработана математическая модель верификации SDL по MSC, позволяющая осуществлять проверки в тех случаях, когда другие известные подходы не работают;
- модель верификации расширена для обработки сложных случаев предметной области;
- создан алгоритм автоматизированной генерации SDL кода по конечному автомату с возможностью улучшения структуры генерируемого кода;
- разработаны алгоритмы по выделению процедур и обработке параллельных участков MSC при генерации SDL кода;
- разработана математическая модель, на базе которой было создано расширение MSC диаграмм. Данное расширение позволяет создавать полное описание поведения объекта в реальных системах.

Эти решения были реализованы, интегрированы с технологией REAL, произведено их использование в реальных промышленных системах.

Литература

[1] Ахо А., Сети Р., Ульман Д. Компиляторы. Принципы, технологии, инструменты. — М.:Вильямс, 2003 — ISBN 5-8459-0189-8, 0-201-10088-6. — 768 с.

[2] Гагарский Р.К. Комплекс средств визуализации для модели параллельных процессов — дипл. р., каф. системного программирования, мат-мех ф-т, СПбГУ — 2004. — 26 с.

[3] Гойхман В.Ю., Гольдштейн Б.С., Дымарский Я.С., Сибирякова Н.Г. Сертификационные испытания коммутационного оборудования средств связи — СПб.: Вестник МАИСУ, 2002. — № 4.

[4] Гольдштейн Б.С. Сигнализация в сетях связи. — М., 1997. — 423 с.

[5] Иванов А.Н., Кознов Д.В., Мурашова Т.С. Поведенческая модель RTST // Записки семинара кафедры системного программирования «CASE-средства RTST++». — СПб., Издательство СПбГУ, 1998. — Вып. 1. — с. 37-49.

[6] Иванов А.Н., Кознов Д.В., Мурашова Т.С., Парфенов В.В., Терехов А.Н. Объектно-ориентированное расширение технологии RTST // Записки семинара кафедры системного программирования «CASE-средства RTST++». — СПб., Издательство СПбГУ, 1998. — Вып. 1. — с. 17-36.

[7] Карабегов А.В., Тер-Микаэлян Т.М. Введение в язык SDL. — М.: Радио и связь, 1993. — 184 с.

[8] Касьянов В.Н. Оптимизирующие преобразования программ. — М.:Наука, 1988. — 336 с.

[9] Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. — СПб.: БХВ Петербург, 2003. — 1104 с.

[10] Кознов Д.В. Визуальное моделирование компонентного программного обеспечения: Дис... канд. физ.-мат. наук — СПбГУ, 2000. — 82 с.

[11] Кузнецов О.П., Адельсон–Вельский Г.М. Дискретная математика для инженера. — М.: Энергоатомиздат, 1988. — 480 с.

[12] Мансуров Н.Н., Майлингова О.Л. Методы формальной спецификации программ: языки MSC и SDL. — Издательство АО “Диалог–МГУ”, 1998. — 125 с.

[13] Мансуров Н. Синтез исполняемых SDL спецификаций по сценарным моделям // Тр. ин-та Системного Программирования — 1999.

[14] Парфенов В.В., Терехов А.Н. RTST — технология программирования встроенных систем реального времени // Системная информатика. — СПб.:1997 — №5. — с. 228-256.

[15] Рейуорд-Смит В. Дж. Теория формальных языков. — М.: Радио и связь, 1988. — 128 с.

[16] Романовский К.Ю., Ивановский Б.Д., Кознов Д.В., Долгов П.С. Обзор нотаций методологии Real // Технический отчет. — 1999.

[17] Соколов В.В. Проверка SDL диаграмм по MSC диаграммам на основе частичной информации // Системное программирование. — СПб.: 2004. — с. 366-390.

[18] Терехов А.Н. RTST — технология программирования встроенных систем реального времени // Записки семинара кафедры системного программирования «CASE-средства RTST++». — СПб., Издательство СПбГУ, 1998. — Вып. 1. — с. 3-17.

[19] Терехов А.Н. и др. REAL: методология и CASE средство разработки информационных систем и ПО систем реального времени // Программирование. — 1999. — №5. — с. 45-51.

[20] Терехов А.Н., Соколов В.В. Новые возможности технологии REAL // Вестн. С.-Петербург. ун-та. — Сер. 10., 2005. — Вып. 1-2. — с. 64-77.

[21] Терехов А.Н., Соколов В.В. Реализация стыка между MSC– и SDL–диаграммами в технологии REAL // Программирование. — 2007. — № 1. — с. 35-50.
Terekhov A., Sokolov V. Implementation of the Conformation of MSC and SDL Diagrams in the REAL Technology // Programming and Computer Software (Engl., Transl.) — 2007. — № 1. — P. 24-33. — DOI: 10.1134/S0361768807010045.

[22] Аннотация средства KLOCWork на сайте SDL Forum.
http://www.sdl-forum.org/Tools/klocwork.htm

[23] Проекты Института Системного Программирования Российской Академии Наук.
http://www.ispras.ru/groups/case/projects.html?2#2

[24] Сайт компании, представляющей средство KLOCWork. http://www.klocwork.com/

[25] Сервер группы OMG, занимающейся развитием методологии UML. Стандарты, публикации, средства. http://www.omg.com

[26] Сервер компании Rational Rose, одной из ведущих компаний средств разработки для методологии UML. http://www.rational.com

[27] Сервер организации по стандартизации ITU-T. http://www.itu-t.org

[28] Concurrent Languages in the Heterogeneous Specification.
http://mint.cs.man.ac.uk/Projects/UPC/Languages/ConcurrentLanguages.html

[29] Use Case Maps (UCMs) нотация. Определения, публикации, средства.
http://www.useCaseMaps.org/index.shtml

[30] Abdalla M., Khendek F., Butler G. New Results on Deriving SDL Specifications from MSCs // Proc. of 9th SDL Forum — 1999. — P. 55-67.

[31] Aho A., Gallagher S., Griffeth N., Scheel C., Swayne D. Sculptor with Chisel: Requirements Engineering for Communications Services // Proc. in 5th International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98) — Lund, Sweden: IOS Press, 1998. — P. 45-63.

[32] Battista G., Eades P., Tamassia R., Tollis I. Algorithms for drawing graphs: An annotated bibliography. // Proc. in Computational Geometry: Theory and Applications 4 — 1994. — P. 235-282.

[33] Ben-Abdallah H., Leue S. Syntactic Analysis of Message Sequence Chart Specifications // Technical Report 96-12 — University of Waterloo — Electrical and Computer Engineering — 1996. — 39 p.

[34] Ben-Abdallah H., Leue S. Syntactic detection of process divergence and nonlocal choice in message sequence charts // Proc. of the Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop (TACAS'97) — №1217 in Lecture Notes in Computer Science — Enschede, The Netherlands: Springer, 1997. — P. 259-274.

[35] Bourduas S., Khendek F., Vincent D. From MSC and UML to SDL // Proc. of IEEE Annual International Conference on Computer Software and Applications (COMPSAC'2002) — Oxford, UK — 2002. — P. 153-158.

[36] Damm W., Harel D. LSCs: Breathing Life into Message Sequence Charts // Proc. in 3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99) — Kluwer Academic Publishers, 1999. — P. 293-312.

[37] Daveau J.-M., Marchioro G., Valderrama C., Jerraya A. VHDL generation from SDL specifications // Proc. of the XIII IFIP Conf. on Com-

puter Hardware Description Languages (CHDL'97) — Toledo, Spain — 1997.

[38] Dulz W., Gruhl S., Lambert L., Söllner M. Early performance prediction of SDL/MSC specified systems by automated synthetic code generation // Proc. of 9th SDL Forum — 1999. — P. 457-472.

[39] EG 201 015 Ver. 1.2.1, Methods for Testing and Specification (MTS); Specification of protocols and services; Validation methodology for standards using Specification and Description Language (SDL); Handbook — 1999. — 28 p.

[40] ETR 184 : Methods for Testing and Specification (MTS); Overview of validation techniques for European Telecommunication Standards (ETSs) containing SDL — 1995. — 27 p.

[41] ETS 300 414, Methods for Testing and Specification (MTS); Use of SDL in European Telecommunication Standards Rules for testability and facilitating validation — 1999. — 99 p.

[42] Elkoutbi M., Khriss, I., Keller R.K. Generating User Interface Prototypes from Scenarios // Proc. in 4th IEEE International Symposium on Requirements Engineering (RE'99) — Limerick, Ireland — 1999. — P. 150-158.

[43] Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion. — 1997. — NASA-GB-001-97. — 245 p.

[44] Holzmann G. Design and Validation of Computer Protocols // Prentice-Hall, 1991 — ISBN 0-13-539834-7. — 512 p.

[45] Horn W., Svantesson B., Kumar S., Jantsch A., Hemani A. Hardware synthesis of an ATM multiplexer from SDL to VHDL: A case study // Proc. of the IEEE Computer Society Workshop On VLSI '99 — Orlando, FL, USA. — 1999. — P. 100-105.

[46] ITU-T MSC2000R3 Draft Z.120: Message Sequence Charts ITU-T Recommendation Z.120. — 1999.

[47] ITU-T Recommendation Z.100: Appendices I and II: SDL Methodology Guidelines, SDL Bibliography. — 1993. — 129 p.

[48] ITU-T Recommendation Z.100: Specification and description language (SDL). — 1999. — 244 p.

[49] ITU-T Recommendation Z.120: Message Sequence Chart (MSC). — 1999. — 136 p.

[50] Khendek F., Vincent D. Enriching SDL Specifications with MSCs // Proc. of 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM 2000) — 2000. — P. 305-319.

[51] Koskimies K., Mäkinen E. Automatic Synthesis of State Machines from Trace Diagrams // Software Practice and Experience, 24(7) — 1994. — P. 643-658.

[52] Svantesson B., Kumar S., Hemani A., A methodology and Algorithms for efficient interprocess communication synthesis from system descriptions in SDL // Proc. of VLSI Design — Chennai, India — 1998. — P. 78-84.

[53] Kurshan R. Computer-Aided Verification of Coordinating Processes // Princeton Series in Computer Science, 1994. — ISBN 0691034362. — 270 p.

[54] Leue S., Mehrmann L., Rezai M. Synthesizing ROOM models from message sequence chart specifications // Proc. in 13th IEEE Conf. on Automated Software Engineering — Honolulu, Hawaii — 1998. — P. 192-195.

[55] Li J., Horgan J. Applying formal description techniques to software architectural design // Computer Communications — Vol. 23 — №12. — 2000. — P. 1169-1178.

[56] Hélouët L., Jard C., Conditions for synthesis of communicating automata from HMSCs // Proc. in 5th International Workshop on Formal

Methods for Industrial Critical Systems (FMICS) — Berlin — 2000. — P. 203-224.

[57] Mansurov N. Automatic Synthesis of SDL from MSC in Forward and Reverse Engineering // Publ. in 1st International Conference on Visual Formal Methods — Eindhoven Technical University, 1999. — P. 44-64.

[58] Mansurov N. Formal methods for accelerated development of telecommunications software // Publ. in Collected Research Papers of the Institute for System Programming — 1999.

[59] Mansurov N., Vasura D. Approximating (H)MSC semantics by Event Automata // Proc. SDL and MSC workshop (SAM 2000) — 2000.

[60] Mansurov N., Zhukov D. Automatic synthesis of SDL models in Use Case Methodology // Proc. of 9th SDL Forum — 1999. — P. 225-240.

[61] Muth A. SDL-based Design of Application Specific Hardware for Hard Real-Time Systems // Dissertation. — Lehrstuhl für Realzeit-Computersysteme, Technische Universität München — 2002. — 208 p.

[62] Robert G., Khendek F., Grogono P. Deriving an SDL specification with a given architecture from a set of MSCs // Proc. of 8th SDL Forum — 1997. — P. 197-212.

[63] Roychoudhury A., Thiagarajan P. Communicating Transaction Processes // Proc. in IEEE International Conference on Application of Concurrency in System Design (ACSD'2003) — 2003. — P. 157-166.

[64] Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual — Addison-Wesley, 1999. — 576 p.

[65] Schönberger S., Keller R., Khriss I. Algorithmic Support for Model Transformation in Object-Oriented Software Development // Proc. in: Theory and Practice of Object Systems (TAPOS). — John Wiley and Sons. — 1999.

[66] Schmid R., Ryser J., Berner S., Glinz M., Reutemann R., Fahr E. A Survey of Simulation Tools for Requirements Engineering // Technical Report 2000.06 — Zurich, Institut für Informatik, 2000.

[67] Schumann J., Whittle J. Automatic Synthesis of Agent Designs in UML // Proc. in 1st Goddard Workshop on Formal Approaches to Agent-Based Systems — NASA Goddard — 2000. — P. 148-162.

[68] Sinclair D., Stone B., Clynch G. An Object Oriented Methodology from Requirements to Validation // Proc. of the 2nd Object Oriented Information Systems Conference (OOIS'95). — Springer, 1995. — P. 265-286.

[69] Somé S., Dssouli R. An Enhancement of Timed Automata generation from Timed Scenarios using Grouped States // The Electronic Journal on Networks and Distributed Processing — 1997.

[70] Somé S., Dssouli R., Vaucher J. From scenarios to timed automata: building specifications from users requirements // Proc. in Asia Pacific Software Engineering Conference (APSEC'95) — 1995. — P. 48-57.

[71] Somé S., Dssouli R., Vaucher J. Toward an Automation of Requirements Engineering using Scenarios //Journal of Computing and Information 2(1) — 1996. — P. 1110-1132.

[72] Telelogic Tau 4.2 documentation, March 2001.

[73] Turner K. Formalising the Chisel Feature Notation // Proc. in 6th International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00) — Glasgow, Scotland, UK : IOS Press, Amsterdam, 2000. — P. 241-256.

[74] Whittle J., Shumann J. Generating Statechart Designs From Scenarios // Proc. in: 22th International Conference on Software Engineering (ICSE 2000) — Limerick, Ireland : ACM Press — 2000. — P. 314-323.

Используемые элементы MSC и SDL диаграмм

MSC и SDL стандарты достаточно объемлющи и различаются в зависимости от года издания, поэтому приведем список элементов, которые используем в данной работе.

A.1 Поддержанный синтаксис MSC

Множество MSC документации делится на диаграммы. На диаграмме изображается набор объектов, в ней участвующих, и взаимодействие объектов. Считается, что временная ось направлена сверху вниз. На MSC диаграммах для нас является существенным следующий набор элементов, приведенный ниже. Полный стандарт MSC описан в [49, 46].

1. Экземпляр объекта. Изображен на рисунке A.1. Наверху в прямоугольнике изображено имя объекта. Картинка, изображенная справа, отличается тем, что на ней объект O1 завершает существование. Слева изображение объекта O1 просто обрывается на диаграмме, при этом объект не завершает существование.

2. Посылка сообщения от объекта O1 объекту O2. Графически изображена на рисунке A.2. Над стрелкой обычно пишется название сообщения.

3. Прием сообщения объектом O1 от объекта O2. Графически изображен на рисунке A.3. Над стрелкой обычно пишется название сообще-

ния.

4. Ссылка на диаграмму. Подразумевается, что эта диаграмма специфицируется в другом месте. Графически изображена на рисунке А.4.

5. Возможность склейки нескольких диаграмм (Condition). Графически изображена на рисунке А.5.

6. Возможность выбора одного варианта поведения из нескольких возможных. Графически изображена на рисунке А.6.

MSC диаграммы часто сопровождаются пояснительными комментариями на естественном языке.

А.2 Поддержанный синтаксис SDL

Для SDL описаний в данной работе будем использовать и текстовый, и графический варианты. Для демонстрации примеров удобнее графический вариант, а при описании алгоритмов генерации будем использовать текстовое описание. Оба описания идентичны друг другу. Полный стандарт SDL описан в [48].

Данная модель обычно содержит вставки на АЯВУ и, в частности, работу с переменными, соответствующими АЯВУ, в который осуществляется генерация. В рамках данной работы этот вопрос не является существенным.

А.2.1 Графическое описание

1. Состояния

   (a) Начальное состояние. Изображено на рисунке А.7.

   (b) Обычное состояние. Изображено на рисунке А.8.

   (c) Завершающее состояние. Изображено на рисунке А.9.

2. Посылка сообщения. Изображена на рисунке А.10.

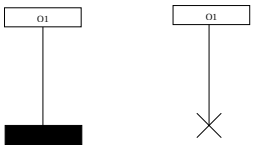

Рис. А.1: Экземпляр объекта на MSC диаграмме с завершением существования (справа) и без (слева)

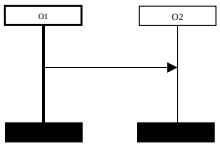

Рис. А.2: Посылка сообщения от объекта O1 объекту O2 на MSC диаграмме

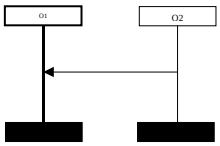

Рис. А.3: Прием сообщения от объекта O2 объектом O1 на MSC диаграмме

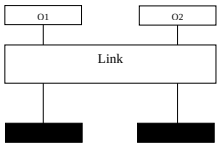

Рис. А.4: Ссылка на другую MSC диаграмму

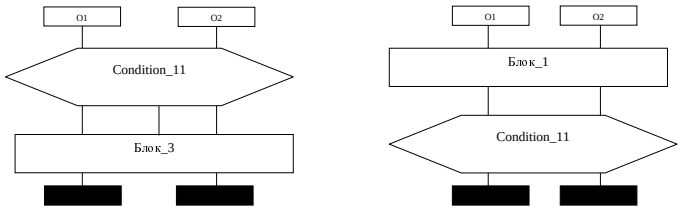

Рис. А.5: Возможность склейки нескольких MSC диаграмм

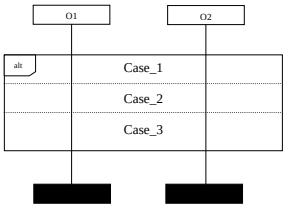

Рис. А.6: Возможность выбора одного варианта поведения из нескольких возможных на MSC диаграмме

Рис. А.7: Начальное состояние SDL

Рис. А.8: Обычное SDL состояние

Рис. А.9: Завершающее состояние SDL

3. Прием сообщения. Изображен на рисунке А.11.

4. Ветвление в коде в зависимости от условия (условий). Изображено на рисунке А.12.

5. Соединитель (Connector). Изображен на рисунке А.13.

6. Соединительные линии. Изображены на рисунке А.14.

Рис. А.10: Посылка сообщения на SDL диаграмме

Рис. А.11: Прием сообщения на SDL диаграмме

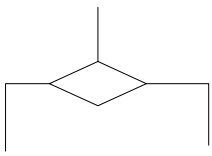

Рис. А.12: Ветвление в SDL коде в зависимости от условия (условий).

Рис. А.13: SDL соединитель

Рис. А.14: Соединительные линии SDL диаграммы

А.2.2 Формат текстового описания

Определим текстовое описание SDL. Для этого приведем пример используемого текстового описания SDL комментариями в С–подобном стиле. Полноценная грамматика SDL дана в стандарте [48].

```
start;                      //Описание стартового состояния
    output Query;           //Посылка сообщения
    nextstate State_1;      //Переход в состояние State_1

state State_1;              //Описание (обычного) состояния State_1
    input Reply;            //Прием сообщения
        join Conn_2;        //Переход на соединитель

state State_3;
    input B1;               //Ниже описываются действия после приема
                            //сообщения B1
        output C1;
        join Conn_2;
    input B2;               //Ниже описываются действия после приема
                            //сообщения B2
        output C2;
        join Conn_2;
```

```
state State_5;
    input Finish_reply;
        stop;               //Переход в завершающее состояние

connection Conn_2;          //Описание соединителя Conn_2
    decision;               //Описание начала ветвления
        decision_thread;    //Описание нити ветвления
            output A;
            nextstate State_3;
        decision_thread;    //Описание другой нити ветвления
            output D;
            nextstate State_4;
        decision_thread;    //И еще одной нити ветвления
            output Finish;
            nextstate State_5;
    enddecision;            //Завершение описания ближайшего ветвления
```

Важное замечание

Использование технологии для создания телефонных станций показало, что использование конструкции Save затрудняет программирование, особенно при долгом сопровождении. Часто возникают проблемы, когда добавился новое сообщение, а затем оно где-то уничтожается из-за неполного описания в конструкции Save. Поэтому было принято решение ко всем состояниям при генерации кода на АЯВУ добавлять оператор сохранения всех сообщений, и не использовать его при создании диаграмм. Следует подразумевать его наличие у всех обычных состояний.

Грамматика предложенного расширения

Приведем грамматическое описание для синтаксического анализатора yacc.

```
%start program

/*
   Keywords used in the program text.
*/
%token K_PROGRAM
%token K_PROGRAM_END
%token K_BEGIN
%token K_END

%token K_CASE
%token K_ESAC
%token K_OF

%token K_IF
%token K_THEN
%token K_ELSE
%token K_FI

%token K_FOR

%token K_WHILE
%token K_DO

%token K_FUNCTION
%token K_PROCEDURE
```

```
%token K_RETURN

%token K_ALT_BEGIN
%token K_ALT_END
%token K_ALT
%token K_OPT_BEGIN
%token K_OPT_END

/*
   Lexical types returned by a scanner.
*/
%token L_DIGIT      /* DIGIT */
%token L_NAME       /* IDENTIFIER */

/*
   Punctuation marks and composite symbols.
*/
%token COMPARE      /* =; >=; <=; >; <; != */

%token SCOLON       /* ;  */
%token COLON        /* :  */
%token COMMA        /* ,  */


%%

/* General file structure */

program         : program_heading
                  procedure_and_function_declaration_list K_PROGRAM_END
                | program_heading K_PROGRAM_END
                ;

program_heading : K_PROGRAM identifier SCOLON
                ;

identifier      : L_NAME
                ;
```

```
procedure_and_function_declaration_list : procedure_or_function_declaration
            | procedure_or_function_declaration
              procedure_and_function_declaration_list
            ;

procedure_or_function_declaration : procedure_declaration
            | function_declaration
            ;

procedure_declaration : procedure_heading code_block
            ;

procedure_heading : K_PROCEDURE identifier SCOLON
            ;

function_declaration : function_heading code_block
            ;

function_heading : K_FUNCTION identifier SCOLON
            ;

/* Block structure */

code_block  : K_BEGIN list_statement K_END
            ;

list_statement : list_statement SCOLON statement
            | statement
            ;

statement   : simple_statement
            | structured_statement
            ;

simple_statement : procedure_statement
            | empty_statement
            | return_statement
            ;
```

```
procedure_statement : procedure_identifier
            ;

procedure_identifier : identifier
            ;

empty_statement : empty
            ;

empty       :
            ;

return_statement : K_RETURN
            | K_RETURN L_DIGIT
            ;
            /* Procedure should return nothing
             * Function should return value
             */

structured_statement : compound_statement
            | conditional_statement
            | repetitive_statement
            | msc_statement
            ;

compound_statement : K_BEGIN list_statement K_END
            ;

conditional_statement : if_statement
            | case_statement
            ;

if_statement : K_IF expression K_THEN statement K_FI
            | K_IF expression K_THEN statement K_ELSE statement K_FI
            ;

expression  : function_call relational_operator L_DIGIT
            ;
```

```
function_call : identifier
            ;

relational_operator : COMPARE
            ;

case_statement : K_CASE function_call K_OF list_case_list_element K_ESAC
            | K_CASE function_call K_OF list_case_list_element
              K_ELSE statement K_ESAC
            ;

list_case_list_element : list_case_list_element SCOLON case_list_element
            | case_list_element
            ;

case_list_element : list_case_label COLON statement
            ;

list_case_label  : list_case_label COMMA L_DIGIT
            | L_DIGIT
            ;

repetitive_statement : while_statement
            | for_statement
            ;

while_statement : K_WHILE expression K_DO statement
            ;

for_statement : for_endless_statement
            | for_number_statement
            ;

for_endless_statement : K_FOR K_DO statement
            ;

for_number_statement : K_FOR L_DIGIT K_DO statement
            ;
```

```
msc_statement : msc_alt_statement
            | msc_opt_statement
            ;

msc_alt_statement : K_ALT_BEGIN statement list_alt_statement K_ALT_END
            | K_ALT_BEGIN statement K_ALT_END
            ;

list_alt_statement : list_alt_statement alt_statement
            | alt_statement
            ;

alt_statement : K_ALT statement
            ;

msc_opt_statement : K_OPT_BEGIN statement K_OPT_END
            ;
```

Список иллюстраций