

**МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА**

Факультет вычислительной математики
и кибернетики

Мансуров Н. Н., Майлингова О. Л.

**Методы формальной
спецификации программ:
языки MSC и SDL**

*(учебное пособие для студентов
4 курса факультета ВМиК)*

1998

УДК 681.142

Пособие посвящено современным объектно-ориентированным методам разработки программного обеспечения. Подробно рассматриваются все фазы процесса разработки, такие как анализ требований, системный анализ, системное проектирование, детальное проектирование и реализация. Описываются техники построения формальных моделей, стандартные языки спецификаций MSC (язык диаграмм взаимодействия) и SDL (язык спецификаций и описаний), а также приводится практическое руководство по системе SDT - инструментальной системе поддержки разработки программного обеспечения на основе языков MSC и SDL. Пособие предназначено для поддержки одноименного лекционного курса.

Рецензенты:

А.Н. Томилин, д.ф.-м.н., профессор

А.К. Петренко, к.ф.-м.н.

Мансуров Н. Н., Майлингова О. Л. Формальные методы спецификации программ: языки MSC и SDL

(учебное пособие для студентов 4 курса факультета ВМиК МГУ).

Издательский отдел факультета ВМиК МГУ

(лицензия ЛР № 040777 от 23.07.96), 1998.- 126 с.

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В.Ломоносова.

ISBN 5-89407-021-X

@Издательский отдел факультета
вычислительной математики и
кибернетики МГУ им. М.В. Ломоносова
1998

ПРЕДИСЛОВИЕ

Настоящее пособие представляет собой практическое руководство по применению формальных методов разработки программного обеспечения. Пособие предназначено для поддержки курса лекций, читаемого авторами на факультете вычислительной математики и кибернетики Московского государственного университета с 1997 года.

Пособие представляет современный процесс разработки программного обеспечения как процесс построения моделей. Основное внимание уделяется двум разновидностям моделей: сценарию использования системы и объектная модель системы. Подробно рассматривается процесс уточнения и детализации моделей на каждой фазе разработки: анализ требований, анализ системы, системное проектирование и детальное проектирование.

Содержание пособия разделяется на три части. Первая часть посвящена методике построения формальных моделей систем на каждой фазе процесса разработки.

Вторая часть пособия посвящена описанию некоторых нотаций, которые используются для формализации моделей. Основное внимание уделяется языку диаграмм взаимодействия MSC. Материал, публикуемый в пособии, представляет собой первую публикацию по данному языку на русском языке. Отдельная глава посвящена языку спецификаций и описаний SDL.

Заключительная часть пособия представляет собой практическое руководство по системе SDT – инструментальной системе поддержки разработки программного обеспечения на базе языков MSC и SDL. Материал практического руководства представляет собой практикум, предназначенный для самостоятельной проработки в терминальном классе рабочих станций.

Авторы выражают благодарность фирме TeleLOGIC AB (Швеция) за предоставление лицензий на систему SDT и содействие в организации курса.

СОДЕРЖАНИЕ

Глава 1. Введение	8
1.1. Фаза анализа требований	9
1.2. Фаза анализа системы	11
1.3. Фаза системного проектирования	12
1.4. Фаза детального проектирования	13
Глава 2. Сценарные модели	14
2.1. Основные понятия	14
2.1.1. Агент	15
2.1.2. Сценарий	15
2.1.3. Интерфейс	15
2.2. Связи между сценариями	15
2.2.1. Отношение использования	15
2.2.2. Отношение расширения	16
2.3. Сценарная модель	17
2.4. Процедура моделирования	17
2.4.1. Описание сценариев использования	18
Глава 3. Архитектурные модели	20
3.1. Устойчивые структуры	21
3.2. Интерфейсные объекты	22
3.3. Информационные объекты	22
3.4. Управляющие объекты	23
3.5. Заключение	24
Глава 4. Язык диаграмм взаимодействия	25
4.1. Основные понятия	25
4.1.1. Диаграмма	25
4.1.2. Объект	26
4.2. События	27
4.2.1. Сообщение	27
4.2.2. Действие	29
4.2.3. Создание объекта	29
4.2.4. Уничтожение объекта	30
4.2.5. Таймер	30
4.2.6. Область неупорядоченных событий	31
4.3. Семантика диаграмм взаимодействия	32
4.4. Структурные средства	35
4.4.1. Состояния	35
4.4.2. Декомпозиция диаграмм	37
4.4.3. Композиция секций	37
4.4.4. Виды композиции диаграмм	38
4.4.5. Иерархическая декомпозиция объектов	39
Глава 5. Язык объектных моделей	42

5.1. Классы	42
5.2. Отношения между классами	42
5.3. Объекты	46
5.4. Модули	47
5.5. Заключение	47
Глава 6. Язык SDL	48
6.1. Теоретическая модель	48
6.2. Графические грамматики	48
6.3. Структура SDL системы	50
6.3.1. Система	53
6.3.2. Блок	53
6.3.3. Процесс	54
6.3.4. Процедура	56
6.4. Взаимодействие	56
6.4.1. Сигнал	57
6.4.2. Список сигналов	58
6.4.3. Канал	58
6.4.4. Межпроцессный канал	59
6.5. Поведение	59
6.5.1. Определение переменных	59
6.5.2. Стартовый символ	59
6.5.3. Состояние	60
6.5.4. Прием сигнала	60
6.5.5. Сохранение сигнала	61
6.5.6. Метки	62
6.5.7. Переход	62
6.5.8. Терминаторы переходов	63
6.5.9. Действия	64
6.5.10. Условия	66
6.5.11. Таймеры	66
6.6. Данные	67
6.7. Заключение	68
Глава 7. Организатор системы SDT	69
7.1. Пример "Игральный автомат"	69
7.2. Запуск системы SDT	69
7.3. Работа с Организатором	70
7.3.1. Окно Организатора	70
7.3.2. Настройка областей Организатора	71
7.4. Работа с деревом SDL системы	72
7.4.1. Добавление корневого узла в дерево системы	72
7.4.2. Сохранение дерева системы	73
7.5. Заключение	74

Глава 8. Редактор диаграмм взаимодействия.....	75
8.1. Создание диаграмм взаимодействия.....	75
8.2. Редактирование диаграмм взаимодействия.....	76
8.3. Заключение	80
Глава 9. Редактор SDL диаграмм.....	81
9.1. Диаграмма системы	81
9.1.1. Создание диаграммы системы	81
9.1.2. Редактирование диаграммы системы.....	83
9.1.3. Добавление новых блоков	84
9.1.4. Перемещение символов и изменение их размеров.....	85
9.1.5. Создание каналов между блоками	85
9.1.6. Создание каналов к окружению	86
9.1.7. Создание текстового символа	86
9.1.8. Сохранение диаграммы	87
9.2. Проверка синтаксиса диаграммы системы.....	88
9.2.1. Запуск Анализатора	88
9.2.2. Где искать сообщения о синтаксических ошибках	90
9.2.3. Как интерпретировать синтаксические ошибки	90
9.3. Диаграммы блоков	91
9.3.1. Создание диаграммы блока из Организатора	91
9.3.2. Создание диаграммы блока по существующей копии.....	93
9.4. Работа с несколькими диаграммами	95
9.5. Диаграммы процессов	96
9.5.1. Создание диаграммы процесса	96
9.5.2. Редактирование диаграммы процесса Demon.....	96
9.5.3. Редактирование процесса Game	100
9.5.4. Редактирование процесса Main	103
9.6. Полный анализ системы	104
9.7. Заключение	105
Глава 10. Выполнение SDL спецификаций.....	106
10.1. Создание исполняемой системы.....	106
10.2. Запуск спецификации на выполнение.....	107
10.3. Выполнение переходов.....	108
10.3.1. Установка уровня детальности трассировки.....	109
10.3.2. Выполнение стартовых переходов системы	109
10.3.3. Посылка сигналов из окружения системы	110
10.3.4. Выполнение переходов.....	112
10.4. Исследование внешнего поведения системы	114
10.4.1. Протокол внешних сигналов.....	114
10.4.2. Добавление новых кнопок к интерфейсу Монитора	115
10.4.3. Взаимодействие с системой.....	116
10.4.4. Просмотр протокола внешних сигналов	117

10.5. Генерация диаграмм взаимодействия	117
10.5.1. Инициализация диаграммы взаимодействия	118
10.5.2. Трассировка	119
10.5.3. Переход к SDL символам	123
10.5.4. Завершение трассировки	124
10.6. Заключение	124

Глава 1. Введение

Настоящий курс посвящен практическому применению формальных методов разработки программного обеспечения. В курсе принята так называемая прагматическая точка зрения на формальные методы, которая заключается в том, что формальные методы применяются, в первую очередь, для достижения точности описаний разрабатываемой системы и, в меньшей степени, для автоматического вывода и доказательства свойств системы. В рамках прагматического подхода, формальные спецификации представляют собой форму документирования понимания системы.

Важно помнить, что в современном процессе разработки сложной программной системы участвует большое количество специалистов, причем одновременно над одним проектом могут работать несколько географически распределенных групп. Кроме того, разработка предполагает проведение разнообразных видов деятельности, требующих взаимодействия специалистов различного профиля. Это означает, что окончательный продукт (т.е. исходные тексты системы, тесты, вспомогательные программы сборки и установки и т.д.) отделены от исходного замысла системы большим количеством моделей. Отсюда следует актуальность задачи документирования моделей, т.е. воплощения моделей в некоторую форму для сохранения, распространения среди коллектива разработчиков, а также для анализа и верификации этих моделей.

Назначение метода разработки (формального, строгого, или структурного) заключается, в первую очередь, в определении конкретной формы моделей и последовательность их построения. В настоящее время общепризнана так называемая фазированная разработка, основанная на построении формальных спецификаций и описаний системы.

Мы будем рассматривать процесс разработки программного обеспечения, состоящий из пяти основных видов деятельности (фаз):

Анализ требований

Анализ системы

Системное проектирование

Детальное проектирование

Реализация

Назначение фазы анализа требований (requirements analysis) состоит в исследовании предметной области, для которой разрабатывается система, а также в уточнении требований к системе. Основной особенностью фазы анализа требований является точка зрения на системы как на «черный ящик», находящийся в некотором окружении. Основные усилия разработчиков направляются на уточнение непосредственного окружения системы.

Назначение фазы анализа системы (system analysis) заключается в нахождении основных объектов системы, необходимых для достижения требуемой функциональности.

Назначение фазы системного проектирования (system design) заключается в уточнении архитектуры системы, выделении компонентов системы (подсистем) и определении детальных интерфейсов между компонентами системы. Фаза системного проектирования включает в себя также дополнительный анализ системы с точки зрения среды разработки и нахождение такой организации компонентов системы, при которой возможна последующая реализация несколькими независимыми группами разработчиков.

Фаза детального проектирования (detailed design) заключается в уточнении поведения всех объектов системы.

Фаза реализации (implementation) заключается в создании исполнимого приложения, отвечающего исходным требованиям.

Интересно, что на ранних фазах разработки имеет место преимущественно аналитическая деятельность, связанная с обнаружением и описанием объектов и свойств, тогда как на более поздних фазах имеет место преимущественно синтетическая деятельность, связанная с изобретением, построением новых объектов и свойств.

Мы будем рассматривать современный объектно-ориентированный метод разработки. Форма моделей и последовательность их построения представлена на Рис. 1. Основу метода составляют модели двух типов: объектные модели и сценарные модели.

Рассмотрим фазы разработки и соответствующие модели более подробно.

1.1. Фаза анализа требований

Фаза анализа требований разделяется на два вида деятельности:

Анализ предметной области

Анализ требований к системе.

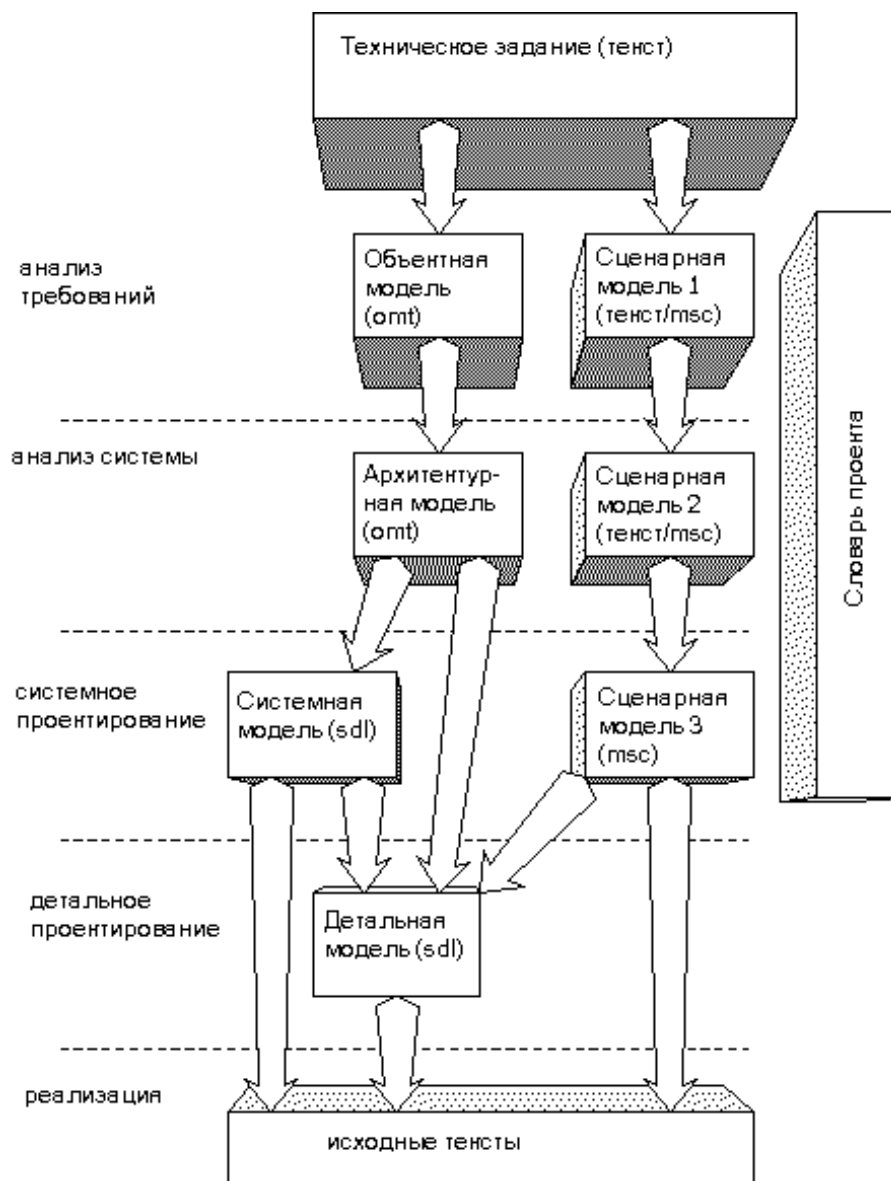


Рис. 1. Фазы разработки и модели

Назначением анализа предметной области является достижение понимания среды, для которой разрабатывается система, безотносительно к самой системе. Формой представления знания о предметной области является так называемая объектная модель требований (объектная модель на Рис. 1). Объектная модель документирует все понятия предметной области, а также связи (отношения) между этими понятиями. Преимущества построения такой модели трудно переоценить:

Разработчики и заказчики получают общее понятийное поле, которое можно использовать для проверки взаимопонимания

Объектная модель существенно облегчает построение архитектурной модели. Объектная модель позволяет быстро интегрировать новых членов коллектива разработчиков, ранее не знакомых с данной предметной областью.

Для формализации объектной модели мы будем использовать так называемый язык объектных моделей (ОМТ).

Назначением анализа требований к системе является формализация требований заказчика к разрабатываемой системе. Формой представления знания о решаемой задаче является так называемая сценарная модель. Сценарная модель представляет собой набор описаний сценариев использования системы, т.е. набор целенаправленных обменов информацией между пользователем и системой, ведущих к достижению пользователями одной из своих целей (ради достижения которых и разрабатывается система!). Преимуществом сценарного подхода к моделированию внешних требований к системе является его ориентация на пользователя, что дает возможность заказчикам оценивать правильность понимания разработчиками задачи.

Для формализации сценарных моделей мы будем использовать язык диаграмм взаимодействия MSC.

Словарь проекта представляет собой простое, но эффективное средство в арсенале разработчиков. Словарь проекта представляет собой определения всех понятий, обнаруженных в предметной области, а также понятий, вводимых при анализе. Назначением словаря проекта является стандартизация терминологии, применяемой всеми участниками проекта, включая разработчиков, заказчиков и пользователей.

1.2. Фаза анализа системы

Рассмотрим теперь следующую фазу – анализ системы. Назначением фазы анализа системы является выявление архитектуры разрабатываемой системы. Основу системного анализа составляет поиск оптимального набора объектов, взаимодействие которых обеспечит выполнение требований к системе. Таким образом, на данной фазе происходит анализ того, какую информацию необходимо представлять в системе (данные, алгоритмы, интерфейсы.), а также распределение функциональности по компонентам системы. Источником информации является преимущественно объектная модель требований и сценарная модель требований. Формой документирования результатов системного анализа является так называемая архитектурная модель – объектная модель, описывающая компоненты системы, их назначение (т.е. выполняемые им функции) и их взаимосвязи.

Именно на этапе анализа системы происходит выбор стабильной, устойчивой к изменениям, расширяемой архитектуры, которая будет способна обеспечить последующую эволюцию системы на протяжении ее ожидаемой жизни.

Дополнительно, на фазе системного анализа происходит уточнение сценариев использования системы и описание динамики архитектуры, т.е. описание взаимодействия компонентов системы по каждому сценарию. Формой документирования динамики архитектуры являются диаграммы взаимодействия MSC.

1.3. Фаза системного проектирования

На данной фазе принимаются окончательные решения по одному из самых важных вопросов разработки – определение архитектуры системы, т.е. того, как сложная система строится из более простых частей, которые, в свою очередь, могут состоять из еще более простых частей и так далее. Выявление архитектуры происходит на фазе анализа системы, но ее точное определение является основным содержанием фазы системного проектирования.

Источником информации для фазы системного проектирования являются архитектурная модель и соответствующая ей сценарная модель, построенные на фазе анализа системы. Результатом фазы является детальное описание компонентов системы и их интерфейсов.

На фазе системного проектирования создаются следующие модели:

Определение архитектуры системы

Структура проектных модулей системы

Детальная сценарная модель, соответствующая архитектуре

Архитектура системы описывается на языке SDL. В языке SDL основной структурной единицей является блок. Блок является контейнером для других объектов.

Архитектура описывает логическую структуру системы. Дополнительно, производится разделение системы на проектные модули. Проектные модули представляют собой организационное деление системы на части, которые могут быть реализованы независимо и параллельно друг другу разными группами разработчиков. На языке SDL проектные модули обычно представляются как пакеты. При описании структуры проектных модулей может быть описано, каким образом в системе планируется использовать стандартные каркасы, компоненты и инструментальные средства. Дополнительно здесь принимаются решения по стратегии реализации каждого модуля. Структура проектных модулей описывается при помощи объектной диаграммы.

Интерфейсы компонентов архитектуры системы описываются на языке SDL. В SDL средством описания интерфейса являются определения сигналов. Динамический аспект интерфейсов, т.е. описание взаимодействия блоков, моделируется при помощи очередного уточнения сценарной модели.

1.4. Фаза детального проектирования

Данная фаза завершает разработку системы. Существует тесная связь между данной фазой и предшествующей фазой системного проектирования, т.к. на фазе детального проектирования решаются вопросы представления и определения поведения объектов, тогда как на фазе системного проектирования происходит распределение объектов по блокам и организация коммуникации между блоками.

Глава 2. Сценарные модели

Сценарные модели предназначены для описания внешнего поведения системы. Сценарная модель представляет собой систематическое описание способов использования системы. Техника использования оказывается весьма эффективной при анализе требований заказчиков и пользователей. Преимуществом данной техники является то, что она полностью ориентирована на пользователя системы, т.е. все описания ведутся исключительно с точки зрения пользователя системы. Это означает, что набор сценариев использования системы может использоваться на всех последующих фазах разработки программной системы для верификации принимаемых проектных решений с точки зрения использования системы.

Первая публикация, содержащая достаточно подробное описание метода, появилась в 1992 году. В ней предложена основная терминология: сценарий использования (*use case*), сценарная модель (*use case model*).

В настоящее время техника получила широкое распространение и используется (в той или иной форме) в большинстве современных методов разработки программного обеспечения.

2.1. Основные понятия

Сценарная модель использует понятия *агент* (от англ. actor), *роль*, *интерфейс* и *сценарий*. Каждый *агент* использует систему для достижения определенной *цели*. Основная идея сценарной модели заключается в описании поведения системы как совокупности *ролей* по отношению к агентам. По определению, роль – это определенная функциональность, которой должна обладать система для достижения одним из агентов одной из его целей. Достижение агентом своей цели предполагает *взаимодействие* между агентом и системой. Система взаимодействует с каждым из агентов в одной из ролей. Каждой роли соответствует отдельный *интерфейс*. Каждая роль описывается как *сценарий* взаимодействия между агентом и системой, т.е. как возможные *последовательности событий*. Такой сценарий называется *сценарием использования* системы, т.к. он полностью определяется некоторой целью агента. Подчеркнем, что последовательность событий, описываемая сценарием использования, не обязательно является единственной.

Вся совокупность сценариев использования (по всем ролям системы по отношению ко всем агентам) составляет сценарную модель системы. Рассмотрим основные понятия сценарных моделей более подробно.

2.1.1. Агент

Понятие агент используется для моделирования чего-либо, что существует вне системы и взаимодействует с системой. Агенты существуют *вне* системы. Поэтому поведение агентов не описывается. Взаимодействие агентов между собой обычно также не описывается. Поведение агентов считается недетерминированным.

Обычно агент использует систему для достижения некоторой своей цели и, как правило, является *активным*, т.е. инициирует взаимодействие. Вместе с тем, некоторые агенты могут предоставлять услуги системе. Такие агенты, как правило, являются *пассивными*.

Следует различать понятия агент и *пользователь*. Понятие *агент* моделирует абстрактную роль, которую может играть некоторый *пользователь*, тогда как понятие *пользователь* моделирует конкретный объект. Вообще говоря, один и тот же пользователь может играть разные роли в разное время.

2.1.2. Сценарий

Конкретный экземпляр агента (т.е. пользователь) выполняет определенную последовательность *взаимодействий* с системой. Взаимодействие пользователя с системой имеет определенную цель, достижению которой служит последовательность обменов информацией с системой. По определению, целенаправленная последовательность взаимодействий пользователя с системой называется *сценарием*.

2.1.3. Интерфейс

Понятие *интерфейс* является вспомогательным в технике ролевого анализа. По определению, каждой роли соответствует некоторый интерфейс. Интерфейс определяет конкретную форму сообщений между системой и пользователем (выступающим как соответствующий агент). Каждый агент может использовать несколько интерфейсов (к различным ролям).

2.2. Связи между сценариями

2.2.1. Отношение использования

Каждый сценарий представляет собой описание некоторого поведения. При составлении этих описаний может случиться, что два сценария описывают некоторое общее поведение. Это общее поведение может быть

выделено в самостоятельный *абстрактный сценарий*. При этом два других сценария (которые можно называть *конкретными сценариями*) будут находиться в *отношении использования* с этим абстрактным сценарием.

Таким образом, один (конкретный) сценарий использует другой (абстрактный) сценарий, когда часть поведения конкретного сценария описана в абстрактном сценарии. Несколько конкретных сценариев могут использовать один и тот же абстрактный сценарий. Введение абстрактных сценариев позволяет выделять похожие части поведения и описывать их единственным раз. Такие сценарии называются абстрактными, т.к. они не могут исполняться самостоятельно, а только описывают общие части поведения других (абстрактных и конкретных) сценариев.

Отношение использования очень похоже на отношение наследования между классами в объектно-ориентированном подходе. Различие между отношением использования и отношением наследования заключается в том, что отношение использования предполагает более сильные ограничения на упорядочение (возможно даже – чередование) операций абстрактного и конкретного сценариев, а также на то, как конкретный сценарий может расширять поведение, определенное в абстрактном сценарии. Заметим, что в традиционном определении отношения наследования между классами (например в C++, Smalltalk, Java и т.д.) наследуются только определения операций и атрибутов и не накладывается никаких ограничений ни на их упорядоченность, ни на способы переопределения.

Между тем, в случае отношения использования между сценариями конкретный сценарий полностью определяет в какой момент и каким образом использовать абстрактный сценарий. Это означает, что при изменении абстрактного класса необходимо проверять все конкретные классы, которые его используют. Таким образом, конкретные классы зависят от абстрактных классов.

2.2.2. Отношение расширения

Сценарии могут находиться в отношении расширения. Один сценарий *расширяет* некоторый самостоятельный сценарий (называемый в этом случае *базовым сценарием*), если он может быть "вложен" в базовый сценарий, обогащая его новым поведением. В данном определении важно, что (в отличие от абстрактного и конкретного сценариев) базовый сценарий, описывает законченное, достаточно интересное с точки зрения пользователя, поведение. Базовый сценарий не зависит от возможных расширений. Такой подход к расширениям позволяет избежать возрастания сложности описания системы. Расширения сценариев являются удобным средством описания изменений и дополнений к основному поведению системы.

Расширение сценария применяется в следующих ситуациях:

- для моделирования необязательных частей сценария;
- для моделирования исключительных ситуаций;
- для моделирования независимых под-сценариев, выполняемых в некоторых определенных случаях;
- в ситуации, когда различные сценарии могут быть вставлены в некоторый простой "охватывающий" сценарий.

Отношение расширения между сценариями можно рассматривать как прерывание выполнения базового сценария. При этом базовый сценарий не знает о возникновении прерывания. Точка прерывания определяется расширением. При добавлении новых расширений базовый сценарий не изменяется. Вместе с тем, поскольку расширения зависят от базового сценария, при изменении последнего необходимо проверять все расширения.

2.3. Сценарная модель

Обычно, сценарная модель состоит из следующих трех частей:

- список агентов;
- список сценариев использования;
- набор описаний сценариев использования.

Список агентов должен описывать каждого «обнаруженного» агента, его цели (зачем он использует систему или какие услуги он предоставляет системе). Структура окружения системы представляется графически на так называемой контекстной диаграмме (см. ниже).

Список сценариев использования представляет собой краткое описание каждого сценария. Структура сценариев использования и отношения между ними представляются графически на диаграмме сценариев (см. ниже).

Для описания сценариев использования на разных фазах разработки используются различные формы представления.

2.4. Процедура моделирования

- 1) Выделить всех агентов, взаимодействующих с системой; составить список агентов, составить текстовое описание каждого агента;
- 2) Выделить все интерфейсы; составить описание каждого интерфейса; построить контекстную диаграмму;
- 3) Для каждого агента выделить различные цели; составить список сценариев;
- 4) Для каждого сценария составить текстовое описание в соответствии с Табл. 1.

5) Провести формальный анализ полученных сценариев, выделить общие части, определить абстрактные сценарии; определить отношения расширения между сценариями;

6) Построить диаграмму сценариев.

На фазе системного анализа производится более детальный анализ сценариев в соответствии с Табл. 2.

2.4.1. Описание сценариев использования

Для описания сценариев использования применяют унифицированные формы. Форма первичного описания сценария приведена на Табл. 1. Такая форма удобна на фазе анализа требований. Детальная форма для описания сценария использования приведена на Табл. 2. Такая форма удобна на фазе анализа системы. Формализация сценариев использования проводится на языке диаграмм взаимодействия.

Идентификатор:	Например, UC_1
Название:	Название должно отражать суть данного способа использования системы с точки зрения пользователя
Список агентов:	Внешние агенты, взаимодействующие с системой при выполнении данного сценария (обычно, один из агентов является основным)
Назначение:	Зачем нужен данный сценарий; какие события инициируют данный сценарий; основные варианты развития сценария; что является результатом сценария. Обзор сценария объемом 3-5 строк текста.
Описание:	Детальное описание основной последовательности событий, включая инициирующие события, и результат сценария.
Альтернативы:	Описание «вариаций» основной последовательности, почему они происходят, что является результатом сценария в каждом случае.

Табл. 1. Первичная форма сценария использования

Идентификатор:	Например, UC_1		
Название:	Название должно отражать суть данного способа использования системы с точки зрения пользователя		
Описание:	Краткое описание функциональности системы, определяемой данным сценарием		
Агенты:	Перечислить внешних агентов (включая пассивных), которые участвуют в выполнении данного сценария		
Предусловия:	Перечислить условия, которые должны быть выполнены для того, чтобы данный сценарий мог состояться. Можно сослаться на другие сценарии, условия на систему или на агентов.		
Последовательность событий и реакций			
Код:	Требование:	Событие:	Реакция:
Начинать с ER_1	Код требования	Описание (внешнего) события, например, «когда система получает сообщение X»	Описание желаемой реакции системы на данное событие, по возможности единственное действие.
ER_2			
ER_n			Завершающее действие, которое дает окончательный результат сценария и «закрывает» сценарий
Альтернативы			
AER_x, Где x принимает значение от 1 до n	Код требования	Описание события, если оно отличается от события в ER_x или описание условия	Реакция системы на событие

Табл. 2. Детальная форма сценария использования

Глава 3. Архитектурные модели

Архитектурная модель описывает структуру системы, не зависящую от конкретной среды реализации. Целью архитектурного анализа является обнаружение *логической структуры системы*, такой, что она будет устойчивой и надежной, удобной для сопровождения и последующего расширения.

Архитектурная модель описывает систему с трех точек зрения: *информация, поведение и представление*. Информационная точка зрения описывает информацию, находящуюся в системе, т.е. внутреннее состояние системы. Поведенческая точка зрения описывает изменения внутреннего состояния системы (когда и каким образом система изменяет состояние). Презентационная точка зрения описывает особенности представления системы для внешнего мира.

В информационном пространстве архитектурных моделей выделим «оси» информации, поведения и представления. Построение архитектурной модели представляет собой определение и размещение объектов в данном информационном пространстве. Одной из возможных стратегий является размещение объектов строго вдоль каждой из осей. Такой подход является характерным для *функциональной декомпозиции*, когда функции размещаются вдоль оси поведения, а данные - вдоль оси информации. Однако при таком подходе к проектированию мы получим структуру, неустойчивую к изменениям: изменение информации, в большинстве случаев, является причиной изменения поведения. Таким образом, было бы желательно, чтобы объекты могли объединять в себе и поведенческие, и информационные аспекты, а в некоторых случаях и презентационные аспекты.

Большинство объектно-ориентированных методов анализа используют некоторый единственный универсальный тип объекта, который может использоваться в любой области пространства. Однако и такой подход не гарантирует устойчивость системы к изменениям архитектурных моделей. Для достижения большей устойчивости к изменениям, мы будем рассматривать стратегию, использующую объекты трех различных типов. Архитектурная модель описывает систему с помощью объектов трех типов: *интерфейсные* объекты, *информационные* объекты и *управляющие* объекты. Каждый тип объектов охватывает два, или все три измерения, однако, каждый тип объекта моделирует определенную точку зрения. Информационные объекты моделируют данные, время жизни которых обычно больше времени жизни соответствующего сценария использования. Все поведение, связанное с такого рода данными, должно быть определено в

самих информационных объектах. В свою очередь, интерфейсный объект моделирует поведение и информацию, которые непосредственно поддерживают взаимодействие системы с окружением. Управляющие объекты моделируют функциональность, не связанную напрямую ни с каким другим объектом, например, поведение, связанное с обработкой нескольких элементов данных, или выполнение вычислений и возвращение результата интерфейсному объекту. Хотя такое поведение и может быть определено в объекте любого из двух других типов (поскольку объекты других типов также способны моделировать поведение), оно на самом деле не принадлежит ни одному конкретному информационному объекту, ни одному конкретному интерфейсному объекту.

3.1. Устойчивые структуры

Почему мы считаем, что структура системы, построенная из объектов трех типов, является устойчивой? Основным утверждением является необходимость изменений системы и ее сопровождения. В этом смысле можно определить устойчивую структуру как такую, при которой все изменения оказываются локальными, т.е. изменения касаются не более одного объекта системы. Рассмотрим наиболее частые типы изменения системы. Обычно, чаще всего изменяются функциональность системы и ее интерфейс. Изменения интерфейса системы должны обычно затрагивать только интерфейсные объекты. Изменения функциональности значительно сложнее т.к. функциональность может быть определена в объекте любого типа. Как правило, функциональность, связывающую несколько сценариев, определяют в объекте управления.

Разработка модели анализа состоит в распределении поведения, описанного в сценариях использования, по объектам архитектурной модели. При этом один и тот же объект может использоваться в нескольких различных сценариях. Основной задачей фазы анализа системы является точное определение того, какой объект выполняет какое поведение сценария использования. Сказанное выше не означает, что уже на данном этапе поведение должно быть разбито на отдельные операции, хотя это и возможно. Более адекватным является текстовое описание обязанностей каждого объекта (ролей объектов).

Рассмотрим каждый из трех типов объектов более подробно и опишем подход выделения данных объектов из сценариев использования.

3.2. Интерфейсные объекты

Вся функциональность сценария использования, которая непосредственно зависит от окружения системы, определяется в интерфейсных объектах. Внешние агенты используют объекты данного типа для взаимодействия с системой. Назначением интерфейсного объекта является перевод информации, получаемой от агентов в форму сигналов внутри системы, а также для перевода сигналов системы во внешнее представление для передачи агентам. Иными словами, интерфейсные объекты описывают двунаправленное взаимодействие между системой и внешними агентами (окружением).

Обычно, интерфейсные объекты легко выделить. Существуют, по крайней мере, три стратегии. Интерфейсные объекты могут быть выделены из описания внешних интерфейсов системы. Другой стратегией выделения интерфейсных объектов является анализ сценариев использования и выделение функциональности, зависящей от окружения системы. Более прямой путь к выделению интерфейсных объектов является анализ агентов. Каждый агент должен иметь, по крайней мере, один интерфейс для взаимодействия с системой. Имея выделенные интерфейсные объекты несложно изменять интерфейс системы. Существует два типа интерфейсных объектов: объекты, описывающие интерфейс с другими системами и объекты, поддерживающие интерфейс с пользователем. Взаимодействие межсистемных интерфейсных объектов описывается, как правило, в форме протоколов обмена.

3.3. Информационные объекты

Информационные объекты используются для моделирования данных, время жизни которых больше, чем время выполнения отдельного сценария использования. Очевидно, что именно информационные объекты обеспечивают координацию выполнения различных сценариев использования.

Большинство информационных объектов можно выделить из модели проблемной области. Но не все. Информационные объекты часто бывают связаны с понятиями реального мира, не имеющего отношения к разрабатываемой системе. Таким образом, наиболее важной задачей является моделирование только необходимых информационных объектов. Поэтому информационные объекты нужно выделять непосредственно из сценариев использования. В разрабатываемую модель надо включать только те

объекты, которые необходимы для выполнения сценария. Для хранения информации объекты используют атрибуты. Любой информационный объект может иметь несколько атрибутов. Каждый атрибут имеет тип, который может представлять собой как простой, так и сложный тип данных. Атрибут описывается как связь с именем и кардинальностью. Объекты всех типов могут иметь атрибуты для описания хранимой информации.

Информационные объекты и атрибуты имеют много схожих свойств. Возникает проблема, как отличить атрибут от информационного объекта. Отличие состоит в использовании информации. Информация, которая может обрабатываться самостоятельно, представляет собой информационный объект. Информация, которая тесно связана с другой информацией и никогда отдельно не используется, представляет собой атрибут некоторого информационного объекта. Другими словами, решающим фактором является способ обработки информации в сценарии использования. Некоторая информация может быть представлена информационным объектом одной системы и атрибутом объекта другой системы. Если она используется из различных мест и для различных целей, то представляет собой отдельный информационный объект.

Как правило, не представляет трудности выделить необходимые информационные объекты. Сложнее определить операции и атрибуты данных объектов. Определен единственный способ доступа к информационным объектам – посредством операций над ним. Поэтому набор операций должен быть полным для любого возможного использования объекта. Подробное описание сценария использования является наиболее значимым источником для определения операций. В сценариях использования описаны все события, которые могут происходить. Выделяя части, относящиеся к рассматриваемому информационному объекту, находим требуемый набор операций.

Взаимодействие между информационными объектами осуществляется с помощью связей по взаимодействию.

3.4. Управляющие объекты

В сложных сценариях использования некоторое поведение не желательно описывать с помощью интерфейсных и информационных объектов, поскольку оно напрямую не связано ни с каким из объектов в отдельности. Такое поведение определяется в управляющих объектах.

Управляющие объекты обеспечивают координацию нескольких объектов, совместно реализующих некоторый сценарий использования. Они существуют только во время существования сценария использования.

Как правило, управляющие объекты обнаруживаются по сценарию использования, например можно ввести один управляющий объект на каждый конкретный и абстрактный сценарий. По каждому сценарию использования, как правило, определяются интерфейсные и информационные объекты. Поведение, которое не было определено этими объектами, определяется в управляющих объектах. Некоторые отклонения от указанного подхода могут возникать в случаях, когда в сценарии не остается неопределенного поведения (и управляющий объект не нужен), или когда не определено слишком сложное поведение (и его следует определить более чем одним управляющим объектом). Надо стремиться связывать один управляющий объект с одним агентом. В таком случае упрощается локализация изменений в программе, вызванных изменениями агентов. *Типичными примерами функциональности управляющих объектов являются: поведение, связанное с взаимодействием (обменом), специфические управляющие последовательности, связанные с одним или несколькими сценариями, а также любые действия по связи интерфейсных и информационных объектов.*

3.5. Заключение

В данной главе мы привели основные понятия архитектурных моделей. Было дано определение архитектуры, устойчивой к изменениям и показано, как можно обнаружить такие архитектуры, систематически выделяя объекты трех типов: интерфейсные, информационные и управляющие.

Глава 4. Язык диаграмм взаимодействия

Язык диаграмм взаимодействия (Message Sequence Charts, MSC) - это язык описания поведения системы в виде последовательности событий. События могут относиться к отдельным компонентам системы, к взаимодействиям между компонентами системы либо к взаимодействию между системой и ее окружением. Основное назначение диаграмм взаимодействия – описание последовательностей допустимых взаимодействий между компонентами системы и системой и ее окружением.

Разновидности диаграмм взаимодействия используются при разработке систем реального времени с 60х годов. Особое распространение диаграммы взаимодействия получили в области разработки телекоммуникационных систем. Язык диаграмм взаимодействия стандартизован в 1992 г. Международным Телекоммуникационным Союзом (Рекомендация Z.120 1992). В настоящее время принята новая, значительно расширенная версия стандарта (Рекомендация Z.120 1996). В лекциях язык диаграмм взаимодействия описывается в рамках стандарта 1992 г. Некоторые уточнения семантики диаграмм взаимодействия предлагаются нами.

4.1. Основные понятия

Диаграмма взаимодействия описывает последовательности событий, происходящих с набором объектов (системой взаимосвязанных компонентов). Дополнительно, каждая система рассматривается как *открытая*, т.е. подразумевается наличие некоторого *окружения* системы, с которым система *взаимодействует*.

Основным понятием диаграммы взаимодействий является *трасса* объекта. Для каждого объекта на диаграмме имеется отдельная вертикальная ось. На этой оси откладываются события, имеющие отношение к данному объекту. Считается, что все объекты существуют одновременно и последовательности событий объектов развиваются параллельно.

Основная разновидность события - это взаимодействие двух объектов, либо взаимодействие объекта и окружения системы. Взаимодействие между объектами (а также между объектом и окружением системы) осуществляется только при помощи обмена сообщениями.


4.1.1. Диаграмма

Диаграмма описывает последовательность событий для некоторого множества объектов и его окружения. Окружение системы представляется рамкой диаграммы. Заметим, что иногда (в качестве некоторого соглашения)

окружение системы моделируют при помощи дополнительного объекта с именем ENV (от англ. "environment", т.е. "окружение").

Графический синтаксис:

```
<msc diagram> ::= <msc symbol> contains
    {<msc heading>{<instance area> | <external message area>}*}
<msc symbol> ::= <frame symbol>

<frame symbol> ::= 

<msc heading> ::= msc <msc name>
```

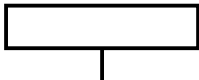
4.1.2. Объект

Каждый объект на диаграмме имеет уникальное *имя*. Дополнительно, для каждого объекта может быть указано, что он является экземпляром некоторого *типа* объектов.

Для соотнесения диаграмм взаимодействия с языком SDL имеется возможность дальнейшего *уточнения типа* объекта: различаются *тип системы* (ключевое слово **system**), *тип блока* (ключевое слово **block**), *тип процесса* (ключевое слово **process**). Уточнение типа объекта имеет смысл только при соотнесении диаграмм взаимодействия с SDL-спецификациями.

Ключевое слово **decomposed** используется для обозначения того, что имеется *иерархическая декомпозиция* данного объекта (диаграмма декомпозиции), на которой трасса данного объекта представлена как диаграмма взаимодействия внутренних компонентов объекта.

Графический синтаксис:


```
<instance area> ::= <instance head area>
    is followed by <instance body area>
<instance head area> ::= <instance head symbol>
    is associated with <instance heading>
<instance heading> ::= <instance name>[:<instance kind> ]
    [decomposed]
<instance head symbol> ::= 

<instance name> ::= <name>
<instance kind> ::= [ <kind denominator> ] <kind name>
<kind denominator> ::= system | block | process
<kind name> ::= <name>
<instance body area> ::= <instance axis symbol>
    {is followed by <instance event area>
    is followed by <instance axis symbol> }*
    is followed by { <instance end symbol> | <stop symbol> }
```



Дополнительно, язык диаграмм взаимодействия позволяет описывать передачу информации в сообщении. С сообщением может быть связан *список параметров*. Каждый параметр моделирует передачу конкретной информации от одного объекта к другому.

Графический синтаксис:

<message out area> ::= <flow line symbol>

<flow line symbol> ::= 

<message in area> ::= <message symbol>
is associated with <msg identification>
is connected to {<message out area>
|<msc symbol>
|<submsc symbol>}
[*is followed by* <message out area>]

<message symbol> ::= 

<external message area> ::= <message symbol>
is associated with <msg identification>
is connected to{<message out area>{<msc symbol>
|<submsc symbol>}}

<msg identification> ::= <message name>

[,<message instance name>] [(<parameter list>)]

<parameter list> ::= <parameter name>[,<parameter list>]

<parameter name> ::= <name>

<message name> ::= <name>

<message instance name> ::= <name>

Дополнительные правила построения диаграмм:

Ограничения на взаимодействие:

1. Событие приема сообщения не должно предшествовать сопряженному с ним событию послылки сообщения (т.е. граф предшествования должен быть ациклическим);
2. Объект не должен посылать сообщений самому себе;
3. Для каждого события послылки сообщения должно быть задано сопряженное с ним событие приема сообщения и наоборот;

Если на трассе объекта область приема некоторого сообщения совпадает с областью послылки другого сообщения (т.е. конец одной стрелки и начало другой находятся в одной точке), то считается, что событие приема сообщения предшествует событию послылки сообщения.

Семантика:

1. Взаимодействие задает частичное отношение порядка на множестве событий всей системы: для данного обмена событие посылки сообщения одного объекта предшествует сопряженному событию приема сообщения другого объекта.
2. События с одинаковыми именами задают взаимодействия одного и того же типа.
3. Семантика параметров сообщения не определяется в языке диаграмм взаимодействия.

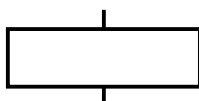
4.2.2. Действие

Дополнительно к описанию взаимодействия объектов, язык диаграмм взаимодействия позволяет описывать действия, выполняемые объектом (например, как реакция на получение некоторого сообщения).

Графический синтаксис:

`<action area> ::= <action symbol> contains <action text>`

`<action symbol> ::=`



`<action text> ::= <text>`

Семантика:

1. Действия локальны по отношению к объекту, который их выполняет.
2. Семантика действия не определяется в рамках диаграмм взаимодействия.
3. Действие происходит "мгновенно".

4.2.3. Создание объекта

Язык диаграмм взаимодействия позволяет описывать динамическое *создание* одних объектов другими объектами. Очевидно, что объект не может взаимодействовать с другими объектами до момента своего создания. При создании объекта ему можно передать некоторую информацию в виде *списка параметров*.

Графический синтаксис:

`<create area> ::= <createline symbol>`

`[is associated with <parameter list>]`

`is connected to <instance head symbol>`

`<createline symbol> ::= - - - - - ►`

Дополнительные правила построения диаграмм:

На данной диаграмме каждый объект может быть создан единственный раз (т.е. к символу заголовка объекта может быть присоединена единственная линия создания объекта).

Семантика:

1. Создание объекта разделяется на два события: одно для создающего объекта и другое для создаваемого объекта;
2. Создание объекта задает отношение порядка между событиями двух объектов: считается, что оба события ("создание" и "рождение") происходят одновременно;
3. Создание объекта означает начало его трассы, т.е. никакие события для данного объекта не могут происходить до его создания;
4. Семантика параметров создания объекта не определяется в языке диаграмм взаимодействия.

4.2.4. Уничтожение объекта

Уничтожение объекта является обратным событием по отношению к созданию.

Графический синтаксис:

`<stop symbol> ::= X`

Семантика:

Уничтожение объекта означает окончательное завершение трассы данного объекта, т.е. никакие события для данного объекта не могут происходить после его уничтожения.

4.2.5. Таймер

Таймер является единственным средством задания времени между отдельными событиями на трассе объекта. Таймер имеет *имя* и, возможно, *длительность*. Каждому таймеру соответствуют два события: *установка таймера* и *срабатывание таймера*. Вместо события срабатывания таймера имеется возможность указать *сброс таймера*.

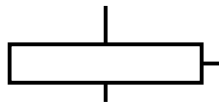
Имеется аналогия между таймерами в языке диаграмм взаимодействия с таймерами в языке SDL. В SDL таймер может быть установлен на определенный промежуток времени, после истечения которого таймер срабатывает и посылает сообщение своему владельцу. Дополнительно, имеется возможность сбросить установленный таймер.

Графический синтаксис:

```
<timer area> ::= <timer set area> | <timer reset area>  
                | <timeout area>
```

```
<timer set area> ::= <set symbol>
```

```
<set symbol> ::=
```



```
<timer reset area> ::= <reset symbol>
```

```
    is associated with <timer name>[( <duration name> )]
```

```
    is connected to <timer set area>
```

```
<reset symbol> ::=
```



```
<timeout area> ::= <timeout symbol>
```

```
    is associated with <timer name>[( <duration name> )]
```

```
    is connected to <timer set area>
```

```
<timeout symbol> ::=
```



```
<duration name> ::= <name>
```

```
<timer name> ::= <name>
```

Дополнительные правила построения диаграмм:

Связь между установкой и срабатыванием таймера:

Для каждого символа установки таймера должен быть задан символ срабатывания, либо символ сброса таймера.

Семантика:

1. Время между установкой и срабатыванием таймера не меньше длительности интервала времени, указанной в символе установки таймера.
2. В случае сброса таймера, время между установкой и сбросом меньше длительности интервала времени, указанной в символе установки таймера.
3. Семантика длительности интервала не определяется в языке диаграмм взаимодействия.

4.2.6. Область неупорядоченных событий

Область неупорядоченных событий служит для описания недетерминированных событий. Например, эта конструкция может быть использована для описания приема двух или более сообщений в произвольном порядке. В языке диаграмм взаимодействия существуют ограничения на события, которые могут происходить внутри области неупорядоченных событий: могут быть только области неупорядоченного приема сообщений и области неупорядоченной отправки сообщений.

Рассмотрим пример диаграммы взаимодействия с именем `ordering`, описывающей систему из трех объекта с именами `a`, `b` и `c`. Типы объектов на диаграмме не указаны.

Объект `a` принимает сообщение с именем `m1` из окружения системы; затем посылает сообщение с именем `m2` объекту `b`; затем посылает сообщение с именем `m3` объекту `c`; затем принимает сообщение с именем `m4` от объекта `b`.

Объект `b` принимает сообщение с именем `m2` от объекта `a`; затем посылает сообщение `m4` объекту `a`.

Объект `c` принимает сообщение `m3` от объекта `a`. Это единственное событие объекта `c` с именем `c` на данной диаграмме.

Ниже приводится графическое представление диаграммы `ordering`:

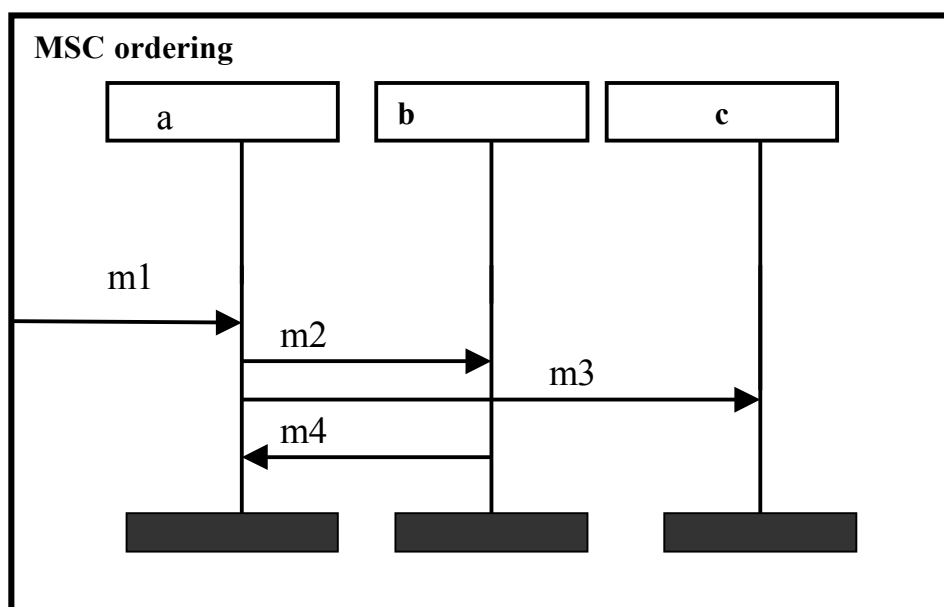


Рис. 2. Диаграмма взаимодействия `ordering`

Рассмотрим текстовое представление диаграммы `ordering`:

```
MSC ordering;
INST a, b, c;
  INSTANCE a;
    IN m1 FROM ENV;
    OUT m2 TO b;
    OUT m3 TO c; *
    IN m4 FROM b;
  ENDINSTANCE;
```

```

INSTANCE b;
    IN m2 FROM a;
    OUT m4 TO a;
ENDINSTANCE;

```

```

INSTANCE c;
IN m3 FROM a;
ENDINSTANCE;

```

ENDMSC;

Обозначим событие приема сообщения m через $in(m)$, а событие посылки сообщения m через $out(m)$.

Множество событий системы на диаграмме `ordering`:

$E = \{in(m1), out(m2), in(m2), out(m3), in(m3), out(m4), in(m4)\}$

Взаимодействия (обмены сообщениями), описанные на диаграмме `ordering`, задают следующие частичные отношения порядка на множестве событий E :

```

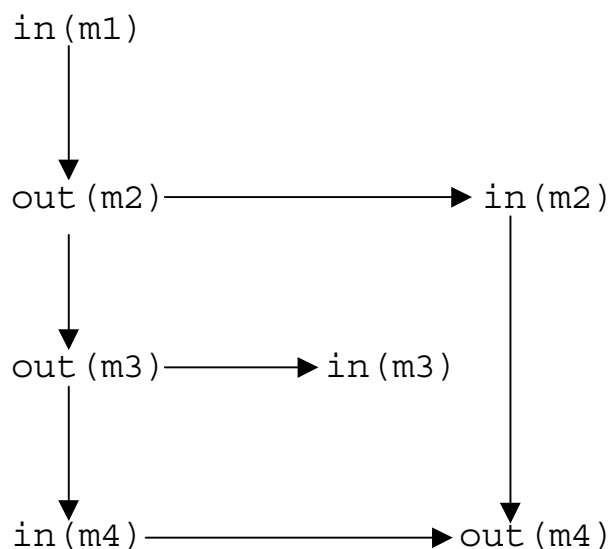
in(m2) < out(m2)
in(m3) < out(m3)
in(m4) < out(m4)

```

Трасса объекта А задает следующее отношение порядка на множестве событий E : $out(m1) < out(m2) < out(m3) < in(m4)$

Трасса объекта В задает следующее отношение порядка на множестве событий E : $in(m2) < out(m4)$

Соответствующее отношение частичного порядка на множестве E можно описать, задав *граф предшествования* (компактная форма представления транзитивного замыкания отношения порядка на множестве E):



Заметим, что любая диаграмма взаимодействия допускает наличие дополнительных наблюдаемых событий между событиями, описанными на диаграмме (в том, числе, наличие дополнительных взаимодействий между объектами).

4.4. Структурные средства

Основным структурным средством языка диаграмм взаимодействия является композиция и декомпозиция посредством состояний. Композиция диаграмм посредством состояний позволяет описывать сложное множество допустимых трасс по частям (т.н. *горизонтальная композиция*).

Дополнительным структурным средством является иерархическая декомпозиция объекта (т.н. *вертикальная декомпозиция*).

4.4.1. Состояния

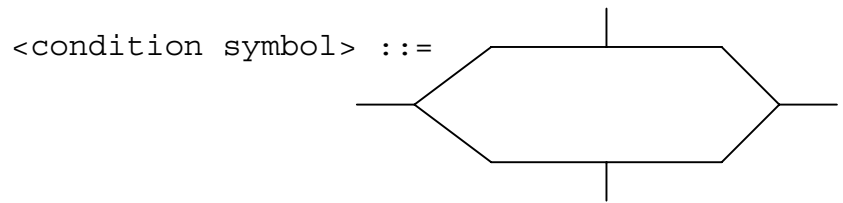
Состояния предназначены для того, чтобы объединять несколько диаграмм для описания сложного поведения множества объектов. Состояние - это особое событие на трассе объекта. В отличие от прочих событий, одно и то же состояние может разделяться одним, двумя и более объектами. По числу объектов на диаграмме, разделяющих некоторое состояние, различают *глобальные* состояния (общее для всех объектов), *разделяемые* состояния (разделяемые несколькими, но не всеми объектами) и *локальные* состояния (разделяемое единственным объектом).

Состояние называется *начальным* (для объекта на диаграмме), если данное состояние предшествует всем остальным событиям на трассе данного объекта. Состояние называется *завершающим* (для объекта на данной диаграмме), если на диаграмме нет событий, которым данное состояние предшествует. Остальные состояния называются *промежуточными* (для объекта на данной диаграмме).

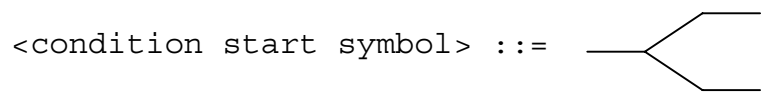
Наибольший интерес для композиции представляют *глобальные* состояния. Глобальное состояние называется *начальным* (на некоторой диаграмме), если оно является начальным для всех объектов на данной диаграмме. Глобальное состояние называется *завершающим* (на диаграмме), если оно является завершающим для всех объектов на диаграмме. Остальные глобальные состояния называются *промежуточными глобальными* состояниями.

Графический синтаксис:

<condition area> ::= <local condition area>
 | <shared condition area>
 <local condition area> ::= <condition symbol>
 contains <condition name>

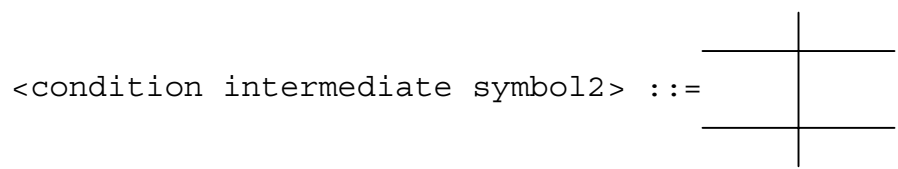
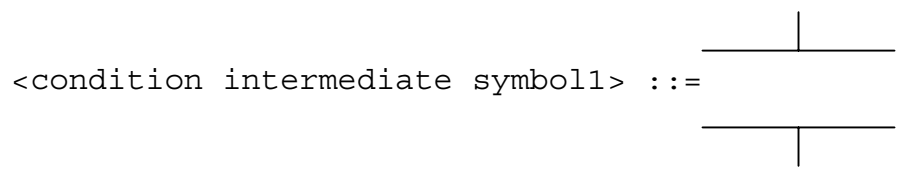


<shared condition area> ::= <condition start area>
 | <condition intermediate area>
 | <condition end area>
 <condition start area> ::= <condition start symbol>
 is associated with <condition name>
 is connected to { <condition intermediate area>
 | <condition end area> }

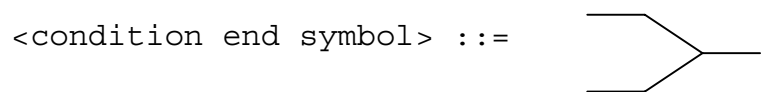


<condition intermediate area> ::= <condition intermediate symbol>
 is connected to { <condition intermediate area>
 | <condition end area> }

<condition intermediate symbol> ::=
 <condition intermediate symbol1>
 | <condition intermediate symbol2>



<condition end area> ::= <condition end symbol>



<condition name> ::= <name>

Дополнительные правила построения диаграмм:

Состояния и сообщения:

Если два объекта разделяют одно и то же состояние, то сопряженные события приема и посылки сообщений должны происходить либо оба до соответствующего состояния, либо оба после соответствующего состояния (т.е. символы передачи сообщений не должны пересекать символы состояний).

Семантика:

1. Состояние является событием для объекта.
2. Разделяемые состояния *не* определяют отношения порядка между событиями различных объектов.
3. Состояния определяют возможные композиции и декомпозиции диаграмм взаимодействия.
4. Семантика диаграммы с состояниями *полностью эквивалентна* семантике данной диаграммы с удаленными состояниями.

4.4.2. Декомпозиция диаграмм

Определим *секцию*, как такую диаграмму, у которой имеются начальное и завершающее глобальные состояния и не имеется промежуточных состояний. Глобальные состояния разбивают диаграмму на секции (т.е. участки трассы между глобальными состояниями).

В общем случае, *декомпозиция* диаграмм посредством состояний определяется следующим образом. Пусть некоторая диаграмма (MSC) содержит промежуточные состояния S_i . Тогда MSC может быть разбита на две диаграммы (MSC1 и MSC2) таким образом, что состояния S_i становятся завершающими на MSC1 и начальными на MSC2.

4.4.3. Композиция секций

Правило *композиции* для секций определяется следующим образом. Пусть имеется две секции, причем множества объектов у них совпадают. Тогда, если на первой секции имеется завершающее глобальное состояние с некоторым именем, а на второй секции- начальное состояния с тем же именем, то вторая секция может служить *продолжением* первой.

Операцию продолжения секции можно уточнить следующим образом: для каждого объекта на новой диаграмме последовательность событий будет состоять из всех событий, описанных на первой диаграмме (в той же

последовательности); затем всех событий, описанных на второй диаграмме (в той же последовательности).

Заметим, что вместе две диаграммы, одна из которых может быть продолжением другой, описывают новое поведение соответствующего множества объектов.

В общем случае, правило композиции диаграмм посредством состояний формулируется следующим образом. Пусть две диаграммы описывают некоторое общее множество объектов I (дополнительно, каждая диаграмма может описывать объекты, отсутствующие на другой диаграмме). Тогда, одна диаграмма ($MSC1$) может служить продолжением другой диаграммы ($MSC2$), если на $MSC1$ объекты из множества I оканчиваются состояниями с именами S_i , и объекты из множества I на $MSC2$ начинаются с соответствующих состояний. При этом некоторые (или даже все) состояния S_i могут быть разделяемыми.

Трассы объектов, не входящие во множество I (на $MSC1$ и $MSC2$), добавляются к объединенной диаграмме. Очевидно, что множество взаимодействий между объектами из $MSC1$ и $MSC2$ должно быть пусто для объектов, не входящих в общее множество I .

Ситуация, когда объект на $MSC1$ посылает сообщение в окружение, а это сообщение на самом деле предназначается одному из объектов на $MSC2$ (т.н. *композиция посредством окружения*) не выражима средствами языка диаграмм взаимодействия.

4.4.4. Виды композиции диаграмм

Рассмотрим основные способы организации набора диаграмм в целях описания сложного поведения множества объектов.

Последовательная композиция

По определению, последовательной композицией называется такая организация набора диаграмм, при которой каждая диаграмма набора (кроме первой) является продолжением некоторой другой (т.е. начальные состояния одной диаграммы совпадают с завершающими состояниями другой диаграммы).

Последовательная композиция описывает сложное поведение, состоящее из нескольких последовательных этапов. Заметим, что поведение, описываемое последовательной композицией нескольких диаграмм можно, вообще говоря, описать единственной диаграммой без состояний.

Альтернативная композиция

По определению, альтернативной композицией называется такая организация набора диаграмм, когда все диаграммы набора (кроме первой) имеют одинаковые начальные состояния, совпадающие с завершающими состояниями первой диаграммы.

Альтернативная композиция описывает сложное поведение, состоящее из взаимоисключающих последовательностей событий. Заметим, что поведение, описываемое альтернативной композицией нескольких диаграмм нельзя, вообще говоря, описать единственной диаграммой без состояний.

Циклическая композиция

По определению, циклической композицией называется такая организация набора диаграмм, когда завершающие состояния одной диаграммы (MSC2) совпадают с начальными состояниями другой диаграммы (MSC1), причем MSC2 является продолжением диаграммы MSC1 (возможно с участием других диаграмм).

Частным случаем циклической композиции является одна диаграмма, которая имеет одинаковые начальные и завершающие состояния.

Циклическая композиция описывает поведение, состоящее из повторяющейся последовательности событий. Заметим, что поведение, описываемое циклической композицией нескольких диаграмм, допускает бесконечное множество конечных последовательностей событий. Такое поведение не может быть описано единственной диаграммой без состояний.

Параллельная композиция

По определению, параллельной композицией называется такая организация диаграмм, когда события одной диаграммы (без изменения последовательности) могут произвольным образом перемешиваться с событиями другой диаграммы.

Параллельная композиция является очень мощным средством описания сложного поведения. Очевидно, что поведение, описываемое параллельной композицией диаграмм нельзя, вообще говоря, описать единственной диаграммой. Параллельная композиция диаграмм *не выражима* средствами стандарта языка диаграмм взаимодействия 1992 г. (но имеется в стандарте 1996 г.).

4.4.5. Иерархическая декомпозиция объектов

Взаимодействие компонентов составного объекта может быть описано отдельно на дополнительной диаграмме (т.н. *диаграмме декомпозиции*). Данное средство поддерживает иерархию описаний поведения сложной системы. Диаграмма декомпозиции отличается от *родительской* диаграммы

взаимодействия только использованием ключевого слова **submsc**. Диаграмма декомпозиции связана с родительской диаграммой, на которой соответствующий объект содержит ключевое слово **decomposed** (имена объекта с ключевым словом **decomposed** на родительской диаграмме и имя диаграммы декомпозиции должны совпадать).

Заметим, что правила связывания объекта с диаграммой декомпозиции приводит к неоднозначности при наличии одноименных объектов с ключевым словом **decomposed** на нескольких диаграммах. Считается, что дополнительная привязка может осуществляться инструментальным средством поддержки.

Для снятия неоднозначности привязки диаграммы декомпозиции к соответствующей основной диаграмме договоримся на каждой диаграмме декомпозиции обязательно указывать имя основной диаграммы. Для этого будем использовать комментарий при заголовке диаграммы декомпозиции со следующим текстом:

```
/* from <parent diagram name> */,
```

где *<parent diagram name>* есть имя родительской диаграммы (с соответствующим ключевым словом **msc** или **submsc**).

Графический синтаксис:

```
<submsc diagram> ::= <submsc symbol>  
    contains {<submsc heading> {<instance area>  
                                |<external message area>}* }
```

```
<submsc symbol> ::= <frame symbol>
```

```
<submsc heading> ::= submsc <submsc name>
```

```
<submsc name> ::= <name>
```

В соответствии с дополнительным соглашением:

```
<submsc heading> ::= submsc <submsc name>
```

```
/* from <parent diagram name> */
```

```
<parent diagram name> ::= msc <msc name> | submsc <submsc name>
```

Дополнительные правила построения диаграмм:

*Связь объекта с ключевым словом **decomposed** и соответствующей диаграммой декомпозиции:*

1. Для каждого объекта, заголовок которого содержит ключевое слово **decomposed**, должна существовать диаграмма декомпозиции с тем же именем.
2. Для каждого события посылки сообщения на трассе объекта, для которого указано ключевое слово **decomposed**, на соответствующей диаграмме декомпозиции должно быть задано событие посылки сообщения в окружение диаграммы декомпозиции.
3. Для каждого события приема сообщения на трассе объекта, для которого указано ключевое слово **decomposed**, на соответствующей диаграмме

декомпозиции должно быть задано событие приема сообщения из окружения диаграммы декомпозиции.

4. Должно существовать соответствие между внешним поведением объекта, для которого указана диаграмма декомпозиции, и его внутренним поведением: последовательность событий объекта на основной диаграмме должен сохраняться на диаграмме декомпозиции (с учетом того, что взаимодействие объекта с другими объектами на основной диаграмме представляется взаимодействием объектов с окружением на диаграмме декомпозиции).

Семантика:

1. Диаграмма декомпозиции уточняет внутреннее поведение некоторого объекта без изменения его внешнего поведения (т.е. описывает взаимодействие компонентов составного объекта между собой и их окружением - объектами на основной диаграмме).
2. Декомпозиция трассы объекта сохраняет порядок приема и отправки сообщений.
3. Действия и состояния на диаграмме декомпозиции *могут* рассматриваться как уточнения действий и состояний на основной диаграмме, однако язык диаграмм взаимодействия *не* предусматривает никаких формальных правил на этот счет.

Глава 5. Язык объектных моделей

При построении объектной модели будем использовать графическую нотацию, позволяющую описывать классы и отношения между классами.

5.1. Классы

Наиболее важным понятием объектной модели является определение *класса*. Классом будем называть описание группы объектов, обладающих общими свойствами и поведением. Все объекты класса имеют одинаковые атрибуты и способны выполнять одинаковый набор операций. Пример определения класса в объектной модели приведен на Рис. 3. Класс `Class1` представлен в так называемой сжатой форме, когда присутствует только имя класса. Класс `Class2` представлен в полной форме. Класс `Class2` имеет два атрибута: `attrib1` (тип не указан) и `attrib2` (типа `atype`). Класс `Class2` имеет две операции: `op1` (аргументы которой не указаны) и `op2` с формальным параметром `arg1` типа `type1` и возвращаемым значением типа `type2`.

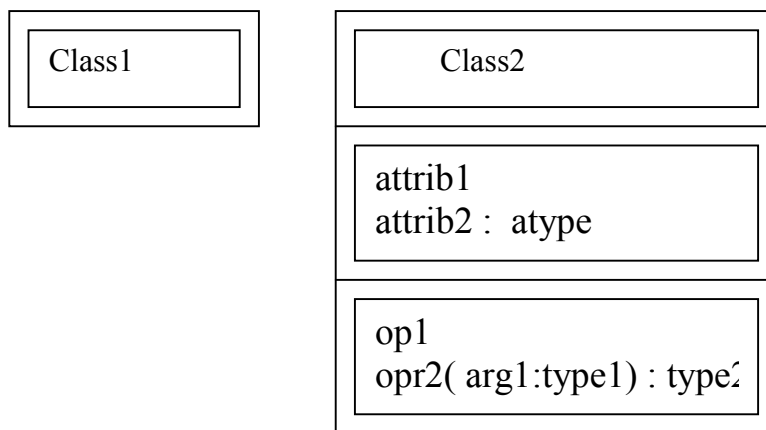


Рис. 3. Определение класса

5.2. Отношения между классами

Классы могут *наследовать* атрибуты и операции других классов. При этом говорят, что классы находятся в отношении наследования. Графическая форма представления наследования показана на Рис. 4.

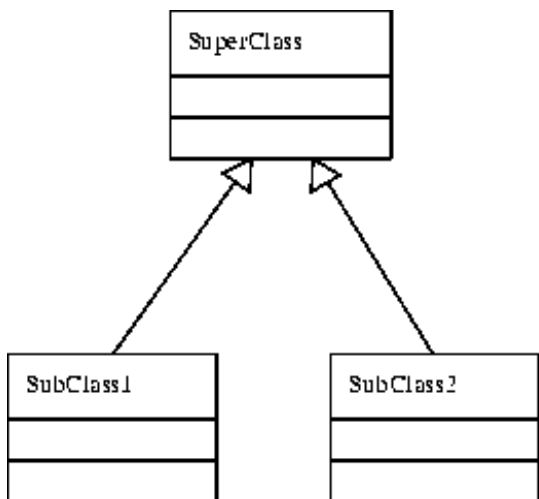


Рис. 4. Наследование

Отношения между классами изображаются графически как связи (*ассоциации*) между классами (Рис. 5). Связь может иметь *имя* и /или может быть помечена *именами ролей* каждого из объектов.

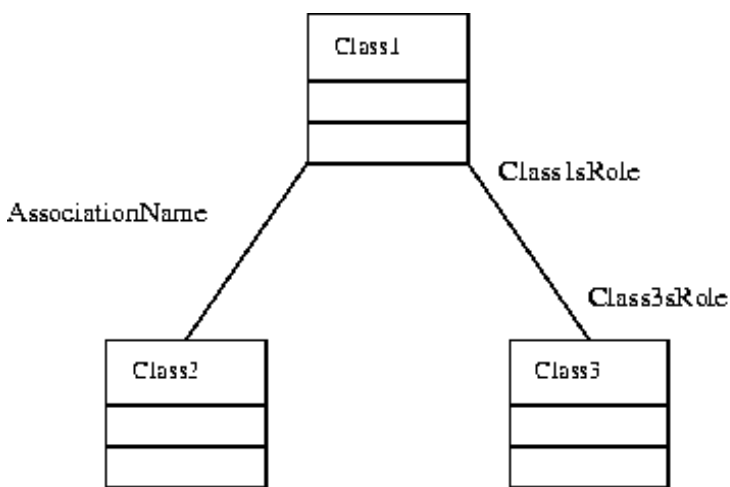


Рис. 5. Ассоциации между классами

Особый интерес представляет отношение «часть – целое». Такое отношение называется *агрегация*. Агрегация классов выделена как особый тип отношений между классами и имеет особое графическое представление, как показано на Рис. 6

При описании ассоциаций и агрегаций основное внимание должно быть уделено выявлению так называемой *кардинальности отношения*, как показано на Рис. 8.

В объектных моделях рассматривают исключительно *бинарные* и *тернарные* отношения. Рассмотрим сначала понятие кардинальности для

бинарных отношений. Рассмотрим пример бинарного отношения (см. Рис. 7)) Отношение имеет имя связан_с. Объекты класса Класс1 выступают в роли левый_партнер, а объекты класса Класс2 – в роли правый_партнер.

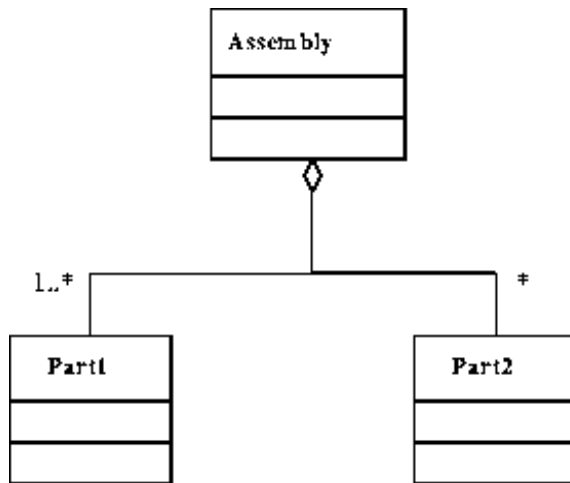


Рис. 6. Агрегация

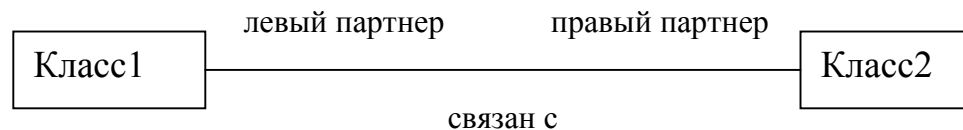


Рис. 7. Пример бинарного отношения

Кардинальность отношения задает, сколько объектов каждого класса может участвовать в отношении. Следует понимать, что отношение между классами является абстракцией систематического отношения между всеми или некоторыми объектами данных классов. Поэтому, дополнительная информация о том, сколько объектов класса могут участвовать в отношении, является существенной.

На диаграмме кардинальность класса задается в виде дополнительного значка на связи возле каждого из определений классов (Рис. 8). Кардинальность класса означает, сколько объектов класс могут вступать с отношение с одним объектом противоположного класса. Кардинальность всего отношения зависит от кардинальности каждого класса.

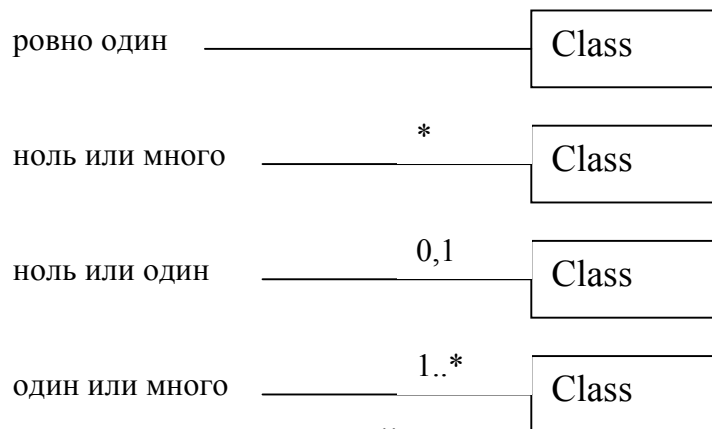


Рис. 8. Кардинальность отношений

Кардинальность класса	Описание
ровно один	С каждым объектом противоположного класса связан ровно один объект данного класса. Это означает, что каждый объект противоположного класса имеет партнера в данном классе (но в данном классе могут быть объекты, не имеющие партнеров из противоположного класса).
ноль или много	С одним объектом противоположного класса могут быть связаны несколько объектов данного класса. Дополнительно, в противоположном классе могут быть объекты, с которыми вообще не связаны объекты данного класса.
ноль или один	С одним объектом противоположного класса может быть связан ровно один объект данного класса, но в противоположном классе могут быть объекты, с которыми не связаны объекты данного класса
один или много	С каждым объектом противоположного класса связано один или более объектов данного класса

Отношения, в которых кардинальность одного из классов содержит ноль («ноль или один», «ноль или много»), называются *условными* отношениями. Отношения, в которых кардинальность обоих классов содержит «много», называются отношением «многие-ко-многим». Обычно для формализации такого отношения вводят дополнительный класс и

превращают отношение в *тернарное*. Пример тернарного отношения приведен на Рис. 9.

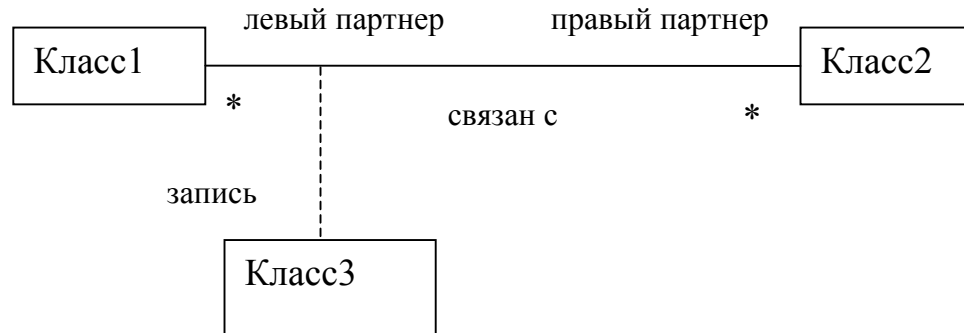


Рис. 9. Пример тернарного отношения

5.3. Объекты

Объектная модель состоит не только из классов, но также из *объектов* и *отношений между объектами*. Отношения между объектами отображаются с помощью связей, аналогично отображению отношений между классами с помощью ассоциаций. Объект изображается в виде прямоугольника с полем, содержащим имя объекта и ссылку на класс, и полем, содержащим имя атрибута и соответствующее ему константное значение или значение, заданное по умолчанию. См. Рис. 10.

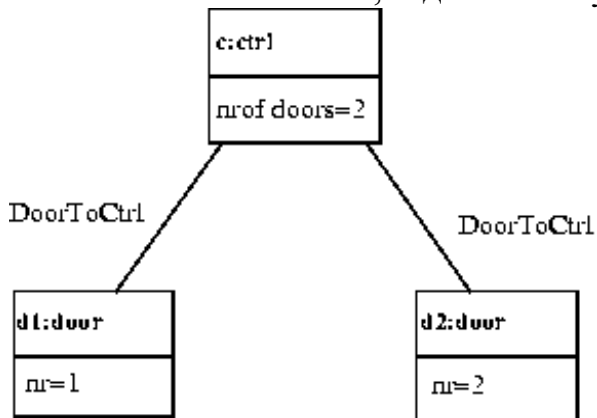


Рис. 10 Объекты и связи между ними

5.4. Модули

Как правило, полная объектная модель состоит из большого количества объектов. Для более наглядного представления объектной модели можно использовать список диаграмм. Необходимо заметить, что для описания одного класса можно использовать несколько диаграмм

Ссылки на классы, определенные в других модулях, префиксируются именем модуля, в котором данный класс определен: ExternalModule::Class.

5.5. Заключение

В данной главе рассмотрена графическая нотация, используемая для описания классов, отношений между классами и отношений между объектами классов. Внимание уделяется рассмотрению кардинальности отношений.

Глава 6. Язык SDL

6.1. Теоретическая модель

SDL система представляет собой набор конечных автоматов, выполняющихся параллельно. Все автоматы независимы и взаимодействуют при помощи послылки дискретных сигналов.

SDL-система состоит из следующих компонентов:

1. Структура - иерархическая декомпозиция, включающая в себя следующие уровни: система, блок, процесс и процедура;
2. Взаимодействие: асинхронные сигналы (возможно с параметрами);
3. Поведение. «Носителями» поведения являются процессы. Поведение процесса описывается как набор состояний, для каждого из которых определены разнообразные реакции на сигналы. Реакция на сигнал (т.н. *переход*) описывается в виде блок-схемы, в которой имеются различные действия, проверки условий и переходы на метки.
4. Данные: алгебраические спецификации абстрактных типов данных; Абстрактный тип данных определяется как *сигнатура* (т.е. набор *литералов* и набор *операторов*), а также набор *алгебраических аксиом*, которым должны удовлетворять операторы. Помимо алгебраических аксиом, имеется возможность определять т.н. *процедурные операторы*. Дополнительно, имеется набор *предопределенных типов данных*. Для работы с данными имеется возможность определять *локальные переменные* в процессах и процедурах;
5. Структурные типы – обобщение структурных единиц в виде типов, определение объектов на основе типов, описание иерархий типов с наследованием и специализацией;

6.2. Графические грамматики

Язык SDL, как и некоторые другие современные языки спецификаций, использует *графическое представление*. Графический синтаксис особенно удобен и дает интуитивное и прозрачное описание поведения системы.

При описании графического синтаксиса возникает проблема формализации описания *пространственных отношений* между символами. Для описания структуры линейного текста, порожденного *операцией конкатенации*, используется аппарат (контекстно-свободных) грамматик. В отличие от линейного текста, диаграмма имеет пространственную структуру. Структура диаграммы порождается *операцией размещения* графических символов на плоскости. Аналогом операции

конкатенации является связывание графических символов, т.е. соприкосновение двух графических символов своими границами на плоскости.

Для описания графического синтаксиса будем использовать аппарат *графических грамматик*. Графическая грамматика состоит из следующих семи частей:

- 1) набор примитивных графических символов;
- 2) описание текстовой лексики для атрибутов примитивных графических символов;
- 3) набор составных графических символов;
- 4) описание атрибутов примитивных графических символов;
- 5) описание составных графических символов;
- 6) описание пространственных связей между графическими символами;
- 7) начальный символ графической грамматики;

Части 1) и 2) соответствуют алфавиту терминальных символов в текстовых грамматиках. Часть 2) описывает правила построения надписей на диаграмме. Часть 3) соответствует алфавиту нетерминальных символов в текстовых грамматиках. Условимся заключать нетерминальные символы в угловые скобки $\langle \rangle$. Части 4), 5) и 6) графической грамматики соответствуют правилам текстовых грамматик. Часть 4) описание атрибутов примитивных графических символов используется для привязки текстовых надписей к графическим символам на диаграмме. По определению, атрибут представляет собой обязательный элемент некоторого нетерминального символа. Обычно, атрибут лишен собственной внутренней структуры, т.е. является терминальным символом. Заметим, что разделение структуры нетерминальных символов на атрибуты и прочие (необязательные) элементы приближает аппарат грамматик к аппарату описания классов в объектно-ориентированном подходе. Часть 6) описание пространственных связей отсутствует в текстовых грамматиках. Часть 7) начальный графический символ соответствует начальному нетерминальному символу в текстовых грамматиках.

Для описания правил построения составных графических символов будем использовать расширенные формы Бэкуса-Наура.

Для описания атрибутов графических символов и пространственных связей между графическими символами дополнительно расширим формы Бэкуса-Наура следующими ключевыми словами:

<i>X is associated with Y</i>	символ Y является атрибутом символа X
<i>X contains Y</i>	X содержит Y
<i>X is connected to Y</i>	X соединен с Y

символы X и Y соприкасаются (обычно боковыми границами)

X is followed by Y

X предшествует Y

специальный случай соединения символов;

X и Y - символы, находящиеся на одной вертикальной оси, причем верхняя граница символа Y соприкасается с нижней границей символа X

6.3. Структура SDL системы

На Рис. 11 показаны четыре основных уровня иерархии в языке SDL: система, блок, процесс и процедура.

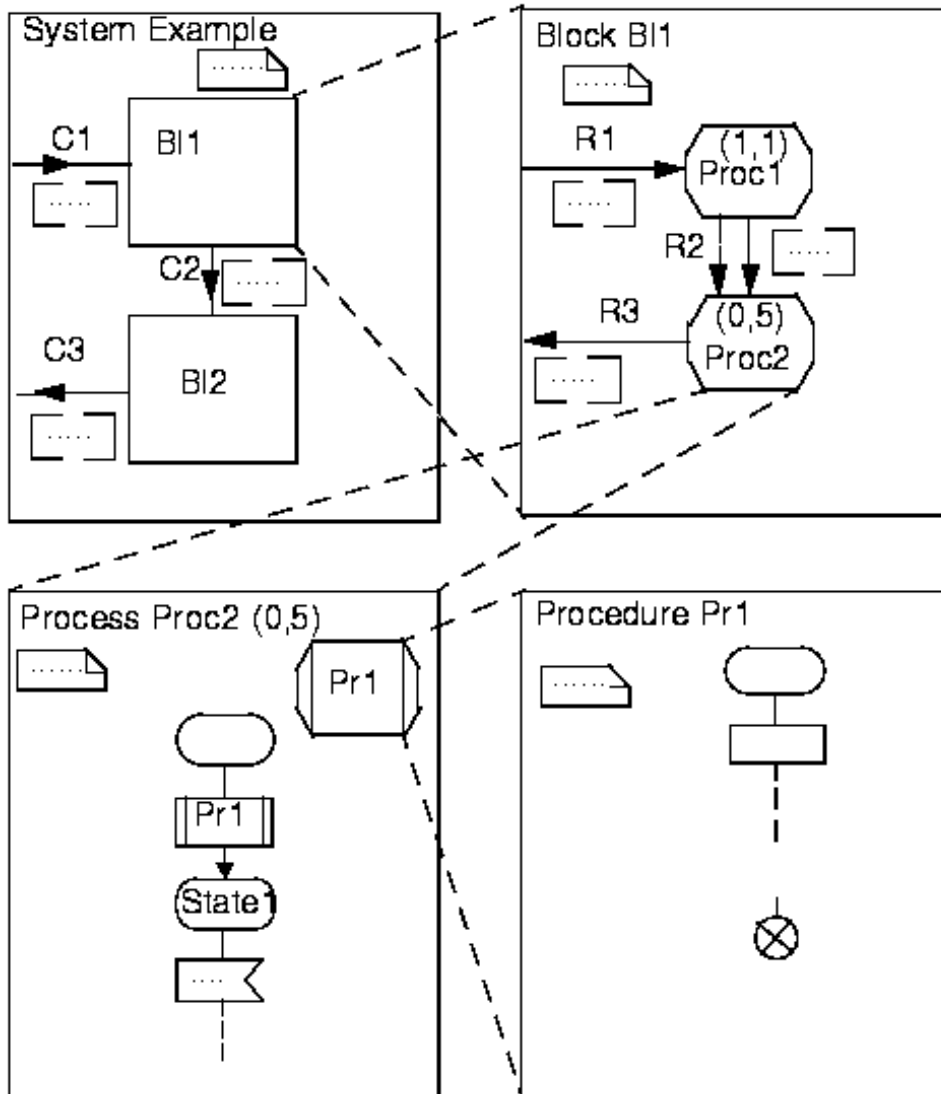


Рис. 11. Средства описания структуры

SDL диаграмма содержит следующие вспомогательные символы:

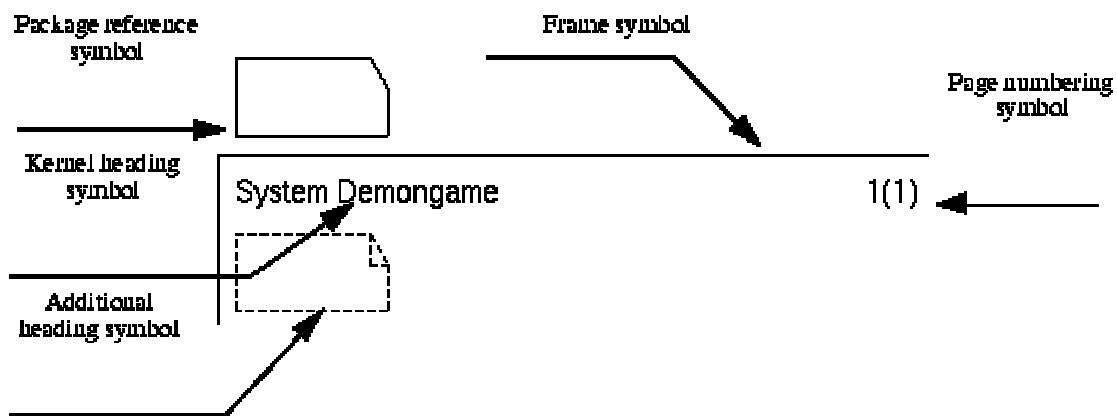


Рис. 12. Вспомогательные символы на диаграмме системы

Символ импорта пакетов

Символ импорта пакетов (Package Reference Symbol) используется для подключения определений из импортируемых пакетов.

Основной заголовок

Основной заголовок (Kernel Heading Symbol) автоматически создается редактором SDL диаграмм. В этом символе содержится информация о типе и имени редактируемой диаграммы. Основной заголовок можно редактировать.

Дополнительный заголовок

Дополнительный заголовок (Additional Heading Symbol) используется для задания наследования и специализации, а также для задания формальных параметров процессов и процедур.

Рамка

Рамка (Frame Symbol) автоматически добавляется редактором. Размер рамки можно изменять, потянув мышью за один из углов.

Номер страницы

Номер страницы (Page Number Symbol) автоматически обновляется редактором. Номер страницы состоит из имени текущей страницы и общего числа страниц. Данный символ нельзя редактировать.

6.3.1. Система

Система представляет собой основной *контейнер* для всех остальных объектов. Система определяет *границу* программного комплекса. Все, что находится вне этой границы, называется *окружением* системы. Внутри системы находится набор *блоков*. Взаимодействие между системой и ее окружением, а также между блоками внутри системы осуществляется только с помощью *сигналов*. Внутри системы, сигналы передаются по *каналам*. Каналы соединяют блоки друг с другом или с окружением системы.

Графический синтаксис:

```
<system diagram> ::=
    [ <package reference area> ] is associated with
        <frame symbol> contains
            { <system heading>
                { <system text area> }*
                <block interaction area>
                { <procedure area>* }
            }
```

<frame symbol> ::=



<system heading> ::= **system** <name>

```
<system text area> ::= <text symbol> contains
    { <signal definition>
      | <data definition> }*
```

```
<block interaction area> ::= {<graphical block reference>
    | <channel definition area> }*
```

```
<graphical block reference> ::= <block symbol> contains <name>
```

<block symbol> ::=



6.3.2. Блок

Блок является *контейнером* для одного или более процессов или содержит *вложенные блоки* (сгруппированные в <block substructure area>). Блок определяет структуру коммуникации для процессов в виде межпроцессных каналов между отдельными процессами, а также между процессами и окружением блока.

Графический синтаксис:

```

<block diagram> ::= <frame symbol>
    contains {<block heading>
        {<block text area> }* <procedure area> }*
        {<process interaction area>|<block substructure area>}}
    is associated with { <channel identifiers> }*
<channel identifiers> ::= <channel identifier>
    {,<channel identifier>}*

```

Примечание:

<channel identifiers> определяют *соединения* межпроцессных каналов с межблочными каналами в окружении блока. Графически, соединения располагаются снаружи рамки блока вблизи конца межпроцессного канала на рамке блока.

```

<block heading> ::= block <name>
<block text area> ::= <system text area>
<process interaction area> ::= { <process area>
    | <create line area>
    | <signal route definition area> }+
<process area> ::= <graphical process reference>
<graphical process reference> ::= <process symbol>
    contains{<process name>[<number of process instances>]}
<process symbol> ::=

```



```

<create line area> ::= <create line symbol>
    is connected to { <process area> <process area> }

```

```

<create line symbol> ::= ----->

```

6.3.3. Процесс

Процесс является основным «носителем» поведения. Несколько экземпляров одного процесса могут работать асинхронно и параллельно друг с другом и экземплярами других процессов.

Каждый экземпляр процесса – это расширенный конечный автомат, взаимодействующий с другими такими же автоматами и с окружением системы. Процесс выполняет последовательности действий (*переходы*) в качестве реакции на получение сигнала. После выполнения цепочки действий процесс *ожидает* получения нового сигнала в некотором состоянии. Каждый процесс имеет

единственный *входной порт* (очередь сигналов), в которую попадают сигналы от всех остальных процессов в порядке поступления.

При порождении системы *статически* создаются некоторые экземпляры процессов. Обмен сигналами начинается только после того момента, когда созданы все статические экземпляры процессов. Остальные экземпляры процессов могут порождаться *динамически* другими процессами (в качестве одного из действий на некотором переходе). При этом на диаграмме блока нужно указывать *связь по порождению* между процессами (от «процесса-отца» к «процессу-сыну»).

Графический синтаксис:

```
<process diagram> ::= <frame symbol>
    contains { <process heading>
                { <process text area> }*
                { <procedure area> }*
                <process graph area> }
<process text area> ::= <text symbol> contains {
    [ <valid input signal set> ]
    { <signal definition>
      | <signal list definition>
      | <variable definition>
      | <data definition>
      | <timer definition> }*
<process heading> ::= process <name>
    [ <number of process instances> ; ]
    [<formal parameters>]
<process graph area> ::=
    <start area> { <state area> | <in-connector area> }*
<number of process instances> ::=
    (<initial number>,<maximum number>)
<initial number> ::= <natural expression>
<maximum number> ::= <natural expression>
<valid input signal set> ::= signalset [<signal list> ] ;
<formal parameters> ::= fpar <parameter of sort>
    { , <parameter of sort> }*
<parameter of sort> ::= <variable name> <sort>
```

Примечание:

<initial number> задает число экземпляров процесса, порождаемых статически при создании системы. <maximum number> задает максимальное число одновременно существующих экземпляров данного процесса.

Конструкция <valid input signal set> позволяет явно специфицировать множество допустимых сигналов процесса (интерфейс процесса). Если эта конструкция не задана явно, интерфейс процесса вычисляется как множество всех входных

сигналов, указанных во всех межпроцессных каналах, присоединенных к данному процессу на соответствующей диаграмме блока.

6.3.4. Процедура

Процедура позволяет объединять набор состояний и переходов в единое целое и вводит некоторое имя для последующей работы с этим набором.

Графический синтаксис:

```

<procedure diagram> ::= <frame symbol>
    contains { <procedure heading>{ <procedure text area>
        | <procedure area> }*<procedure graph area> }
<procedure area> ::= <graphical procedure reference>
<procedure text area> ::= <text symbol> contains
    { <variable definition>
    | <data definition> }*
<graphical procedure reference> ::= <procedure symbol> contains
    { <procedure preamble> <procedure name> }
<procedure symbol> ::=

```



```

<procedure heading> ::= procedure <name>
    [<procedure formal parameters>]
    [<procedure result>]
<procedure graph area> ::= [<procedure start area>]
    { <state area> | <in-connector area> }*
<procedure start area> ::= <procedure start symbol>
    is followed by <transition area>
<procedure start symbol> ::=

```



```

<procedure formal parameters> ::= fpar
    <formal variable parameter>
    { , <formal variable parameter> }*
<formal variable parameter> ::=
    <parameter kind> <parameter of sort>
<parameter kind> ::= [ in/out | in ]
<procedure result> ::= returns <variable name> <sort>

```

6.4. Взаимодействие

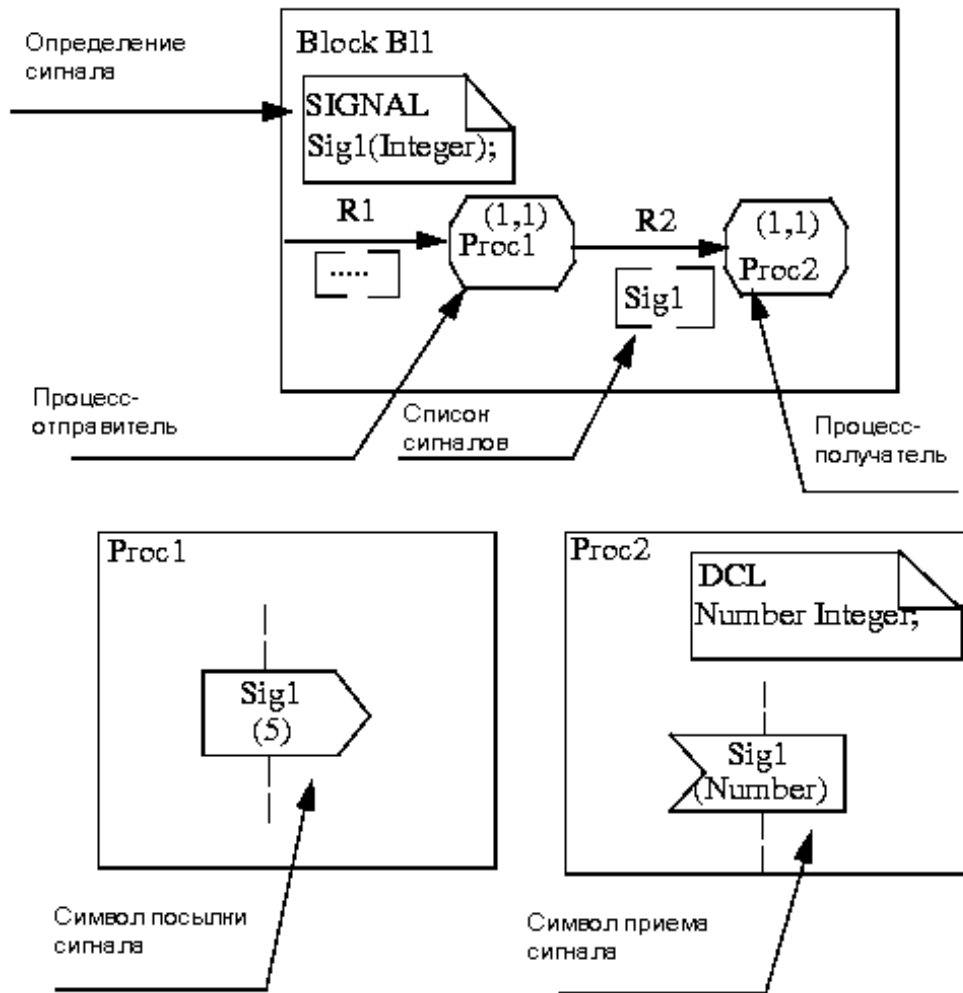


Рис. 13. Посылка сигнала между двумя процессами

В языке SDL нет глобальных данных. Передача информации между процессами или между процессами и окружением системы осуществляется через посылку сигналов (возможно с параметрами). Сигналы передаются от одного процесса (процесса-отправителя) во входной порт другого процесса (процесса-получателя). Процесс-отправитель и процесс-получатель могут совпадать. Посылка сигнала является асинхронной, т.е. процесс-отправитель продолжает работу, не ожидая подтверждения о получении сигнала процессом-получателем. Сигналы передаются по статически определенной сети межблочных и межпроцессных каналов, связанных соединениями. Заметим, что в каждом соединении могут участвовать по несколько каналов как с внешней, так и с внутренней стороны.

6.4.1. Сигнал

Графический синтаксис:

```
<signal definition> ::= signal <name> [<sort list>]  
<sort list> ::= ( <sort> { , <sort> }* )
```

6.4.2. Список сигналов

Список сигналов позволяет объединять несколько сигналов в единую *группу* и использовать имя этой группы в определениях каналов и т.д.

Графический синтаксис:

```
<signal list area> ::= <signal list symbol>  
    contains <signal list>  
<signal list symbol> ::=
```



```
<signal list definition> ::= signallist <name> = <signal list> ;  
<signal list> ::= <signal list item> { , <signal list item> }*  
<signal list item> ::= { <signal name>  
    | ( <signal list name> )  
    | <timer name>  
<signal name> ::= name  
<signal list name> ::= name  
<timer name> ::= <name>
```




6.4.3. Канал

Канал описывает *маршрут* передачи сигналов между двумя блоками или между блоком и его окружением. Каналы могут быть однонаправленные или двунаправленные. С каждым направлением может быть связан свой *список допустимых сигналов*.

Графический синтаксис:

```
<channel definition area> ::= <channel symbol>  
    is associated with { <channel name>  
        <signal list area>  
        [ <signal list area> ] }  
    is connected to { <block area>  
        { <block area> | <frame symbol> }  
<channel symbol> ::= <channel symbol 1>  
    | <channel symbol 2>  
    | <channel symbol 3>
```

```

<channel symbol 1> ::= 
<channel symbol 2> ::= 
<channel symbol 3> ::= 

```

6.4.4. Межпроцессный канал


Графический синтаксис:

```

<signal route definition area> ::= <signal route symbol>
    is associated with <name>
    { <signal list area> [ <signal list area> ] }
    is connected to { <process area>
        { <process area | <frame symbol> }
<signal route symbol> ::= <signal route symbol 1>
    | <signal route symbol 2>


```

```

<signal route symbol 1> ::= 

```

```

<signal route symbol 2> ::= 

```

6.5. Поведение

6.5.1. Определение переменных

Каждый процесс (и процедура) могут статически определять *локальные переменные*. Дополнительно, процесс (и процедура) могут иметь *формальные параметры*. Все переменные создаются при порождении экземпляра процесса (вызове процедуры) и уничтожаются при остановке процесса (возврате из процедуры).

Графический синтаксис:

```

<variable definition> ::= dcl <name> <sort>
    [ := <expression> ] ;

```

Примечание:

При определении переменной можно указать выражение для инициализации переменной при создании.

6.5.2. Стартовый символ

Стартовый символ определяет стартовый переход процесса, т.е. последовательность действий, выполняемую при порождении экземпляра данного процесса (статически или динамически).

Графический синтаксис:

```
<start area> ::= <start symbol>  
                is followed by <transition area>  
<start symbol> ::=
```



6.5.3. Состояние

Состояние определяет набор реакций на сигналы (*переходы*). Каждое состояние имеет *имя*. Состояние с именем «*» является особым: это способ определить переходы, общие для всех обычных состояний.

```
<state area> ::=  
<state symbol> contains <state name> is associated with  
    { <input association area>  
      | <save association area> } *
```

```
<state name> ::= <name> | <asterisk>  
<asterisk> ::= *
```

```
<state symbol> ::=
```



```
<input association area> ::= <solid association symbol>  
                            is connected to  
                            <input area>  
<save association area> ::= <solid association symbol>  
                            is connected to  
                            <save area>
```

6.5.4. Прием сигнала

Символ приема сигнала определяет *реакцию* на данный сигнал (в данном состоянии). Непосредственно при получении сигнала переменные, указанные в символе приема сигнала, получают значения *параметров сигнала*.

Графический синтаксис:

```
<input area> ::= <input symbol> contains { <stimulus> }  
                is followed by <transition area>  
<stimulus> ::= { <signal name> | <timer name> }  
                [ ( <variable> { , <variable> }* ) ]
```

<input symbol> ::=



6.5.5. Сохранение сигнала

Сигналы, указанные в символе сохранения сигнала в некотором состоянии, *игнорируются* процессом в данном состоянии. Эти сигналы остаются во входном порте процесса (в порядке поступления) для последующей обработки в другом состоянии. Наличие механизма сохранения сигналов является важным, потому что в каждом состоянии определена *стандартная реакция* на любой сигнал, которая состоит в том, что сигнал изымается из очереди и уничтожается, тогда как процесс остается в старом состоянии.

Графический синтаксис:

```
<save area> ::= <save symbol> { contains <save list> }  
<save list> ::= { <signal list> | <asterisk> }
```

<save symbol> ::=



6.5.6. Метки

Метки позволяют определять участки переходов, общие для нескольких состояний.

Графический синтаксис:

```
<in-connector area> ::= <in-connector symbol>
                        contains <connector name>
                        is followed by <transition area>
<connector name> ::= <name>
<in-connector symbol> ::=
```



6.5.7. Переход

Переход определяет последовательность действий, выполняемую процессом в качестве реакции на получение некоторого сигнала в некотором состоянии. Во время выполнения перехода процесс может осуществлять доступ к переменным, посылать сигналы, а также порождать экземпляры процессов. Переход заканчивается либо тем, что процесс попадает в новое состояние и ожидает там следующего сигнала, либо остановкой процесса (возврат в случае процедуры), либо передачей управления на другой переход.

Графический синтаксис:

```
<transition area> ::= [<transition string area>] is followed by
                    { <state area>
                      | <nextstate area>
                      | <decision area>
                      | <stop symbol>
                      | <merge area>
                      | <out-connector area>
                      | <return area> }
<transition string area> ::=
    { <task area>
      | <output area>
      | <set area>
      | <reset area>
      | <create request area>
      | <procedure call area> }
    [ is followed by <transition string area> ]
```

6.5.8. Терминаторы переходов

6.5.8.1. Символ перехода в состояние

Символ перехода в новое состояние определяет, в каком состоянии процесс будет ожидать следующего сигнала. `<dash nextstate>` означает, что изменения состояния не происходит. Конструкция `<dash nextstate>` оказывается особенно ценной при организации помеченных участков графа.

Графический синтаксис:

```
<nextstate area> ::= <state symbol> contains <nextstate body>
<nextstate body> ::= { <state name> | <dash nextstate> }
<dash nextstate> ::= -
<state name> ::= <name>
```

6.5.8.2. Символ перехода на метку

Символ перехода на метку осуществляет передачу управления на новый участок перехода. Заметим, что явная передача управления на текущий переход является единственным средством организации циклов.

Графический синтаксис:

```
<merge area> ::= <merge symbol> is connected to
    <flow line symbol>
<merge symbol> ::= <flow line symbol>
<flow line symbol> ::=
```

```
<out-connector area> ::= <out-connector symbol> contains
    <connector name>
<out-connector symbol> ::= <in-connector symbol>
```

6.5.8.3. Символ остановки

Выполнение остановки процесса приводит к уничтожению всех переменных процесса и всех необработанных сигналов из входного порта процесса.

Графический синтаксис:

```
<stop symbol> ::=
```



6.5.8.4. Символ возврата из процедуры

Возврат из процедуры означает продолжение того перехода, на котором произошел вызов процедуры.

Графический синтаксис:

```
<return area> ::= <return symbol>  
                [ is associated with <expression> ]  
<return symbol> ::=
```



6.5.9. Действия

6.5.9.1. Присваивание

Графический синтаксис:

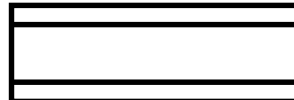
```
<task area> ::= <task symbol> contains <assignment statement>  
<task symbol> ::=
```



6.5.9.2. Создание процесса

Графический синтаксис:

```
<create request area> ::= <create request symbol>  
                        contains <create body>  
<create request body> ::= <name> [ <actual parameters> ]  
<actual parameters> ::= ( <expression> { , <expression> }* )  
<create request symbol> ::=
```



Динамическое поведение SDL системы описывается внутри процессов. Процессы могут создаваться при запуске системы, а также могут создаваться и уничтожаться динамически во время выполнения системы. Может существовать более одного экземпляра одного и того же процесса. Каждый экземпляр процесса имеет уникальный идентификатор PID. Это делает возможным посылку сигнала конкретному экземпляру процесса.

Формальные параметры процесса получают свои значения при его создании. У экземпляров процессов, которые создаются статически, значения формальных параметров неопределены.

У каждого процесса есть четыре стандартные переменные типа Pid:

- Собственный идентификатор процесса (переменная *self*);
- Идентификатор родительского процесса (переменная *parent*);
- Идентификатор последнего процесса, порожденного данным процессом (переменная *child*);
- Идентификатор процесса, от которого был получен последний полученный сигнал (переменная *sender*).

При попытке создания лишних экземпляров процесса (т.е. больше чем `<maximum number>`), новый экземпляр не создается, переменная `offspring` получает значение `Null`, а выполнение перехода продолжается.

6.5.9.3. Вызов процедуры

Графический синтаксис:

```
<procedure call area> ::= <procedure call symbol> contains  
                        <procedure call body>  
<procedure call body> ::= <name> [ <actual parameters> ]  
<procedure call symbol> ::=
```



6.5.9.4. Посылка сигнала

При посылке сигнала можно явно указать маршрут, по которому должен быть направлен сигнал (конструкция `via`), либо процесс-получатель (конструкция `to`) в одной из двух форм: либо имя процесса-получателя, либо уникальный Pid процесса-получателя. Если при посылке сигнала указано имя процесса-получателя, то сигнал попадет к произвольному экземпляру процесса с данным именем (если таковые вообще существуют).

Графический синтаксис:

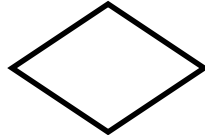
```
<output area> ::= <output symbol> contains <output body>  
<output body> ::= <signal name> [ <actual parameters> ]  
                  [ to <destination> ] [ via <via path> ]  
<destination> ::= <Pid expression> | <name>  
<via path> ::= <channel name>  
               | <signal route name>  
<signal name> ::= <name>  
<channel name> ::= <name>  
<signal route name> ::= <name>  
<output symbol> ::=
```



6.5.10. Условия

Графический синтаксис:

```
<decision area> ::= <decision symbol> contains <question>
                   is followed by
                   { { <graphical answer part>
                       <graphical else part> }*
                     | { <graphical answer part> {<graphical answer part>}+
                       [ <graphical else part> ] }*
<decision symbol> ::=
```



```
<graphical answer part> ::= <flow line symbol>
                            is associated with <graphical answer>
                            is followed by <transition area>
<graphical else part> ::= <flow line symbol> is associated with
                          <else answer>
                          is followed by <transition area>
<graphical answer> ::= <range condition>
<else answer> ::= else
```

6.5.11. Таймеры

Графический синтаксис:

```
<timer definition> ::= timer <timer name> [ <sort list> ]
                    [ := <Duration ground expression> ]
<set area> ::= <task symbol> contains <set>
<reset area> ::= <task symbol> contains <reset>

<set> ::= set <set statement>
<set statement> ::= ( [ <Time expression> ,] <timer name>
                    [ ( <expression list> ) ] )
<reset> ::= reset ( <reset statement> )
<reset statement> ::= <timer name> [ ( <expression list> ) ]
<timer name> ::= <name>
<Time expression> ::= <expression>
<Duration expression> ::= <expression>
```



```

<open range> ::= <constant>
                | { = | /= | < | > | <= | >= } <constant>

<expression> ::= <sub expression>
<sub expression> ::= <operand0> | <sub expression> => <operand0>
<operand0> ::= <operand1> | <operand0> { or | xor } <operand1>
<operand1> ::= <operand2> | <operand1> and <operand2>
<operand2> ::= <operand3>
                | <operand2> { = | /= | > | >= | < | <= | in } <operand3>
<operand3> ::= <operand4> | <operand3> { + | - | // } <operand4>
<operand4> ::= <operand5> | <operand4> { * | / | mod | rem } <operand5>
<operand5> ::= [ - | not ] <primary>
<primary> ::= <name>
                | ( <expression> )

```

6.7. Заключение

В данном разделе мы подробно рассмотрели основные возможности языка спецификаций и описаний SDL.

Мы рассмотрели основные структурные единицы языка SDL: систему, блок, процесс и процедуру. Мы также рассмотрели организацию взаимодействия процессов через послыки сигналов по статическим маршрутам, состоящих из межблочных и межпроцессных каналов, связанных соединениями. Мы рассмотрели средства описания поведения процесса как расширенного конечного автомата.

Отметим, что некоторые возможности языка остались за рамками нашего рассмотрения:

- Структурные типы
 - Определение структурных типов
 - Использование типизированных объектов
 - Наследование структурных типов
 - Специализация структурных типов
 - Специализация поведения процессов
 - Определения пакетов
- Алгебраические спецификации абстрактных типов данных
- Процедурные определения операторов абстрактных типов данных

Глава 7. Организатор системы SDT

Данная часть пособия представляет собой практическое руководство по редактору SDL диаграмм системы SDT. Предполагается, что все упражнения, описанные в данном разделе, будут проделаны на компьютере. В следующих главах мы будем разрабатывать MSC и SDL спецификации примера "Игральный автомат".

7.1. Пример "Игральный автомат"

Внешнее поведение системы выглядит следующим образом. Система воспринимает четыре типа входных сигналов: `Newgame` (новая игра), `Endgame` (конец игры), `Probe` (ход) и `Result` (счет). Первые два сигнала используются для начала и окончания игры. Одновременно может вестись только одна игра, т.е. сигналы `Newgame` игнорируются в течение игры, а сигналы `Endgame` игнорируются, если игра не началась.

Сама игра очень простая. Состояние системы время от времени меняется от выигрышного к проигрышному и обратно. Игрок должен угадать, в какой момент состояние выигрышное. Если игрок делает ход (посылает сигнал `Probe`) в тот момент, когда состояние системы выигрышное, он получает одно очко. Если игрок делает ход в момент, когда состояние системы проигрышное, он теряет одно очко. Игрок может узнать текущий счет, послав сигнал `Result`.

7.2. Запуск системы SDT

Мы предполагаем, что система SDT установлена на компьютере. Полный путь к директории, в которой находятся файлы системы SDT, задан в переменной `$telegologic`. Полный путь к примерам, которые приводятся в упражнениях, задан в переменной `$examples`.

Для работы с руководством нужно создать временную директорию:
`mkdir ~/demongame`

В дальнейшем мы будем предполагать, что имя рабочей директории выглядит именно так.

Для запуска системы SDT нужно сделать следующее:

1. Перейти в рабочую директорию:

`cd ~/demongame`

2. Выполнить команду

`sdt`

7.3. Работа с Организатором

Организатор - это основная среда системы SDT, из которой осуществляется запуск всех остальных инструментов.

После выполнения упражнений данного урока вы должны научиться:

- Настраивать области Организатора;
- Создавать дерево SDL системы в Организаторе;

7.3.1. Окно Организатора

После запуска система SDT открывает окно Организатора (*SDT Organizer*).

Организатор выдает приветственное окно (*SDT Welcome window*). Это окно автоматически исчезает после выполнения любого действия в Организаторе (можно также нажать на кнопку *Continue*).

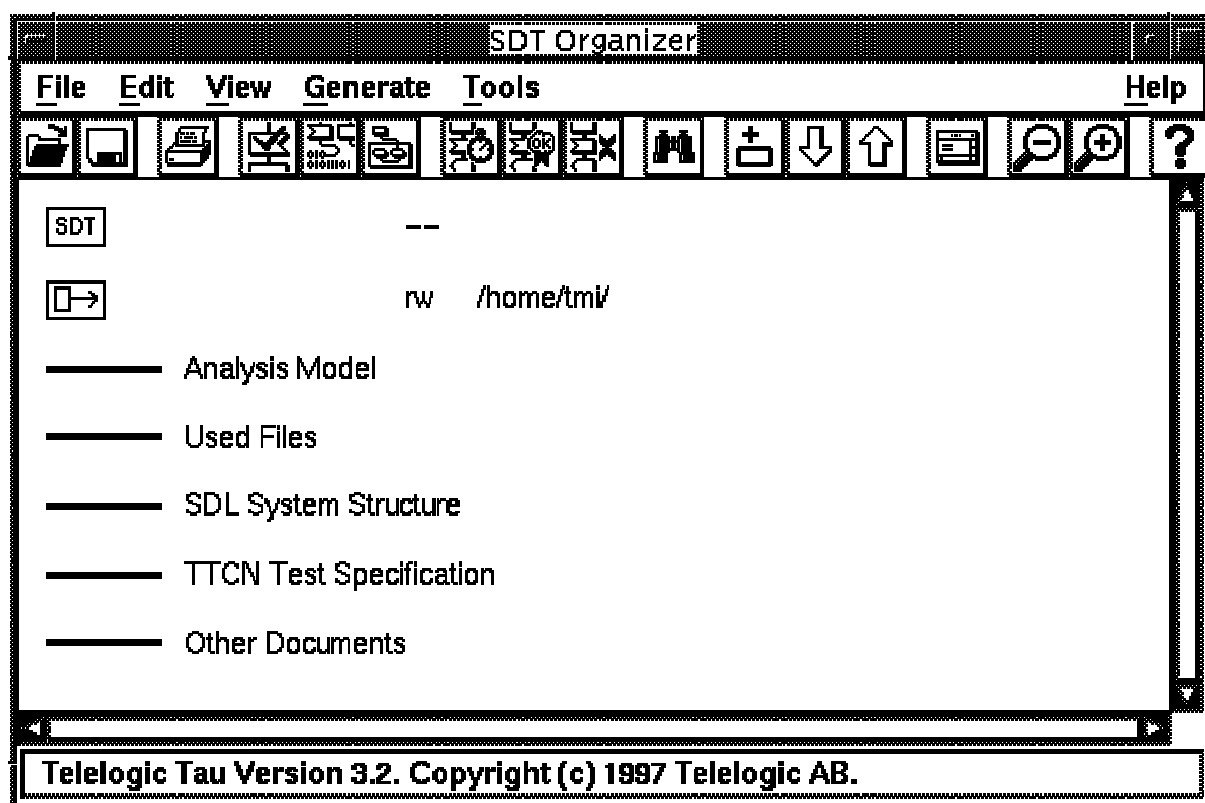


Рис. 15. Окно Организатора

7.3.2. Настройка областей Организатора

После запуска системы Организатор создает 5 стандартных областей (см. Рис. 15):

- Analysis Model («модель анализа»)
- Used Files («используемые файлы»)
- SDL System Structure («дерево системы»)
- TTCN Test Specification («спецификации тестов»)
- Other Documents («прочие документы»)

В каждой из этих областей может находиться несколько диаграмм, использование областей не регламентировано.

В верхней части окна Организатора находятся два символа, соответствующие системному файлу и рабочей директории для диаграмм. Назначение системного файла будет описано ниже. Рабочая директория используется для поиска диаграмм, а также для сохранения диаграмм.

Удалите области Analysis Model, Used Files и TTCN Test Specifications:

1. Выберите область Analysis Model, щелкнув мышью на соответствующий символ в окне Организатора.
2. Выберите команду *Remove* из меню *Edit* Организатора.
3. Подтвердите удаление области Analysis Model в дополнительном диалоге, нажав мышью на кнопку *Remove* (см. Рис. 16).

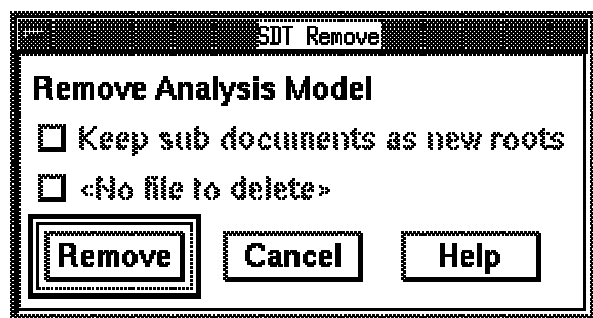


Рис. 16. Подтверждение удаления области

4. Повторите шаги 1-3 для областей Used Files и TTCN Test Specifications.

Переименуйте оставшиеся области:

1. Выберите область SDL System Structure
 2. Выберите команду *Edit* в меню *Edit*.
 3. Введите новое имя области My first SDL system в диалоге *Edit*.
- Внимание: нельзя изменять текущий тип документа (*Organizer*) и его значение (*Area*) (см. Рис. 17).

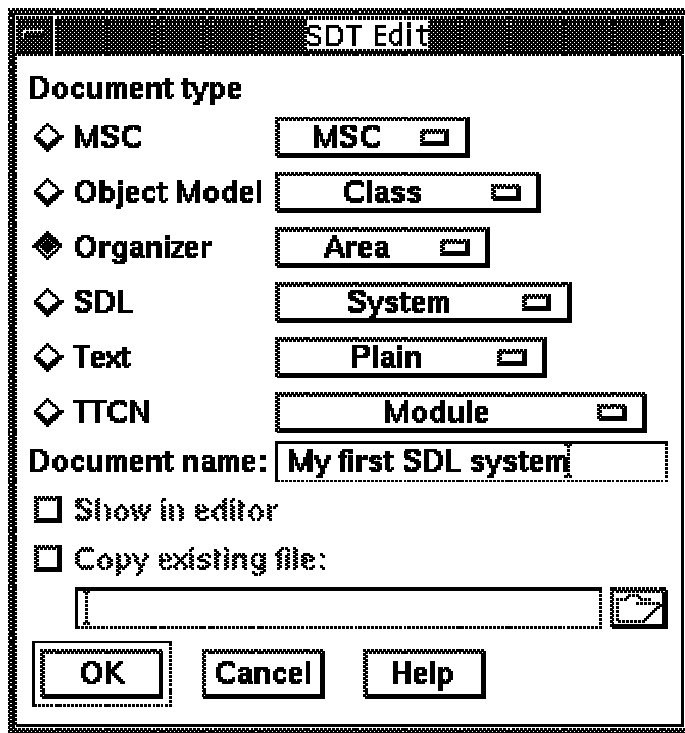


Рис. 17. Переименование области Организатора

4. Завершите выполнение операции, нажав на кнопку *OK*.

7.4. Работа с деревом SDL системы

После выполнения упражнений данного урока вы должны научиться следующему:

- Создавать новую диаграмму системы
- Добавлять новые страницы к диаграмме
- Редактировать диаграмму системы
- Сохранять диаграммы
- Работать с диалогами системы SDT
- Работать с деревом SDL системы

7.4.1. Добавление корневого узла в дерево системы

Для добавления корневого узла в дерево системы нужно проделать следующие операции:

1. Выберите область *My first SDL system* в Организаторе
2. Выберите команду *Add New* из меню *Edit*. Выполнение данной команды приводит к появлению диалога *Add New*, в котором надо задать имя и тип новой диаграммы (Рис. 18).

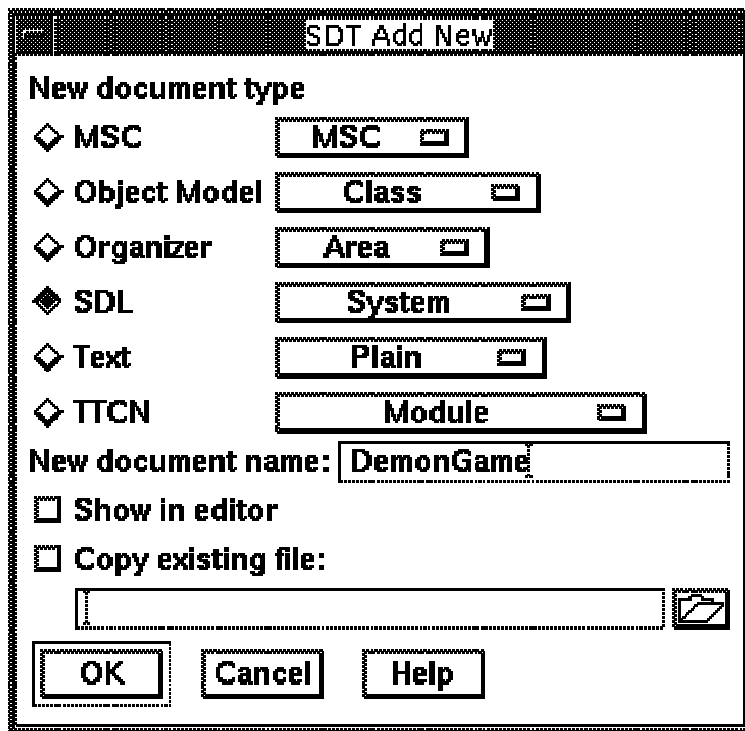


Рис. 18. Добавление новой диаграммы

3. Выберите *SDL* как тип нового документа (колонка *New document type*) и *System* как тип диаграммы в соответствии с тем, как показано выше.
4. Задайте имя диаграммы *DemonGame* (по умолчанию имя новой диаграммы - *Untitled*).
5. Выключите настройку *Show in Editor*.
6. Завершите операцию, нажав кнопку *OK*.

После выполнения операции в окне Организатора появляется корневой узел дерева *SDL* системы - диаграмма системы *DemonGame*. (см. Рис. 19). Диаграмма обозначена как *[unconnected]* (т.е. она пока не связана ни с каким файлом).

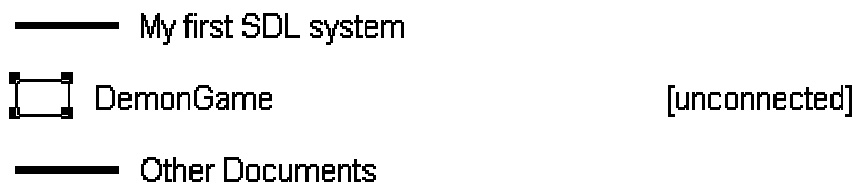


Рис. 19. Корневой узел дерева *SDL* системы

7.4.2. Сохранение дерева системы

Вы сохранили диаграмму системы. Дополнительно, нужно сохранить дерево системы, созданное в Организаторе. Организатор использует для этой цели так называемый системный файл (*System File*).

Для сохранения системного файла нужно выбрать команду *Save* из меню *File* Организатора.

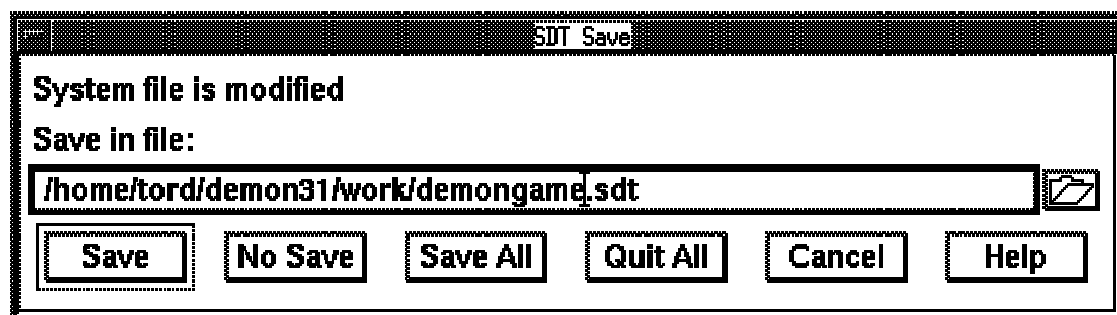


Рис. 20. Диалог Save Организатора

Система SDT предлагает имя системного файла `demongame.sdt` (суффикс `*.sdt` является стандартным для системных файлов). Завершите выполнение команды, нажав на кнопку *OK*.

Команда *Open* из меню *File* Организатора позволяет открыть системный файл уже существующей системы.

Существует возможность сохранить сразу все измененные диаграммы и системный файл при помощи одной единственной команды.

- Нажмите кнопку *Save All* в диалоге сохранения в Организаторе; или
- Нажмите «быструю клавишу» *Save* на инструментальной панели Организатора. Данная команда сохранит все диаграммы и дерево системы без дополнительных подтверждений, кроме случаев, требующих вмешательства пользователя. Заметим, что «быстрая клавиша» *Save* на инструментальной панели редактора сохраняет только текущую диаграмму.

7.5. Заключение

В данной главе были описаны основные приемы работы с Организатором системы SDT. В результате выполнения упражнений данной главы вы должны были научиться:

- настраивать области Организатора;
- создавать дерево SDL системы в Организаторе;
- создавать новую диаграмму системы
- добавлять новые страницы к диаграмме
- редактировать диаграмму системы
- сохранять диаграммы
- работать с диалогами системы SDT
- работать с деревом SDL системы

Глава 8. Редактор диаграмм взаимодействия

Упражнения данного урока позволят вам научиться:

- создавать диаграммы взаимодействия;
- редактировать диаграммы взаимодействия;

8.1. Создание диаграмм взаимодействия

Для создания диаграммы взаимодействия нужно выполнить следующие операции:

1. Выбрать область *Other Documents* в окне Организатора
2. Выбрать команду *Add New* из меню *Edit* (или нажать «горячую клавишу»).
3. Указать тип диаграммы как *MSC*. Ввести имя диаграммы: *DemonGame*. Включить режим *Show in editor*.

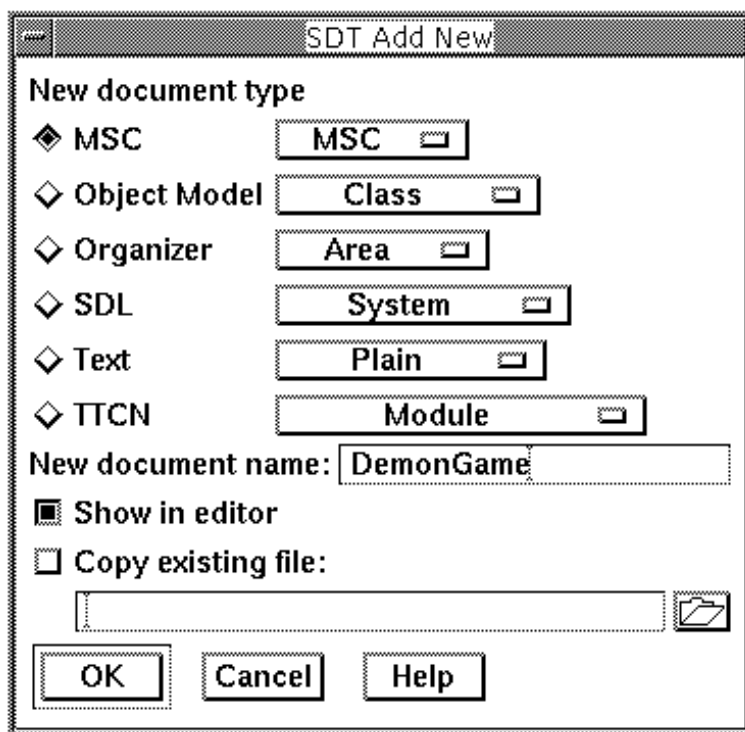


Рис. 21. Задание типа и имени диаграммы взаимодействия

4. Завершите выполнение операции, нажав на кнопку *OK*. В окне Организатора появится символ диаграммы взаимодействия.



Рис. 22. Символ диаграммы взаимодействия

8.2. Редактирование диаграмм взаимодействия

Вид окна редактирования диаграмм взаимодействия приведен на Рис. 23.

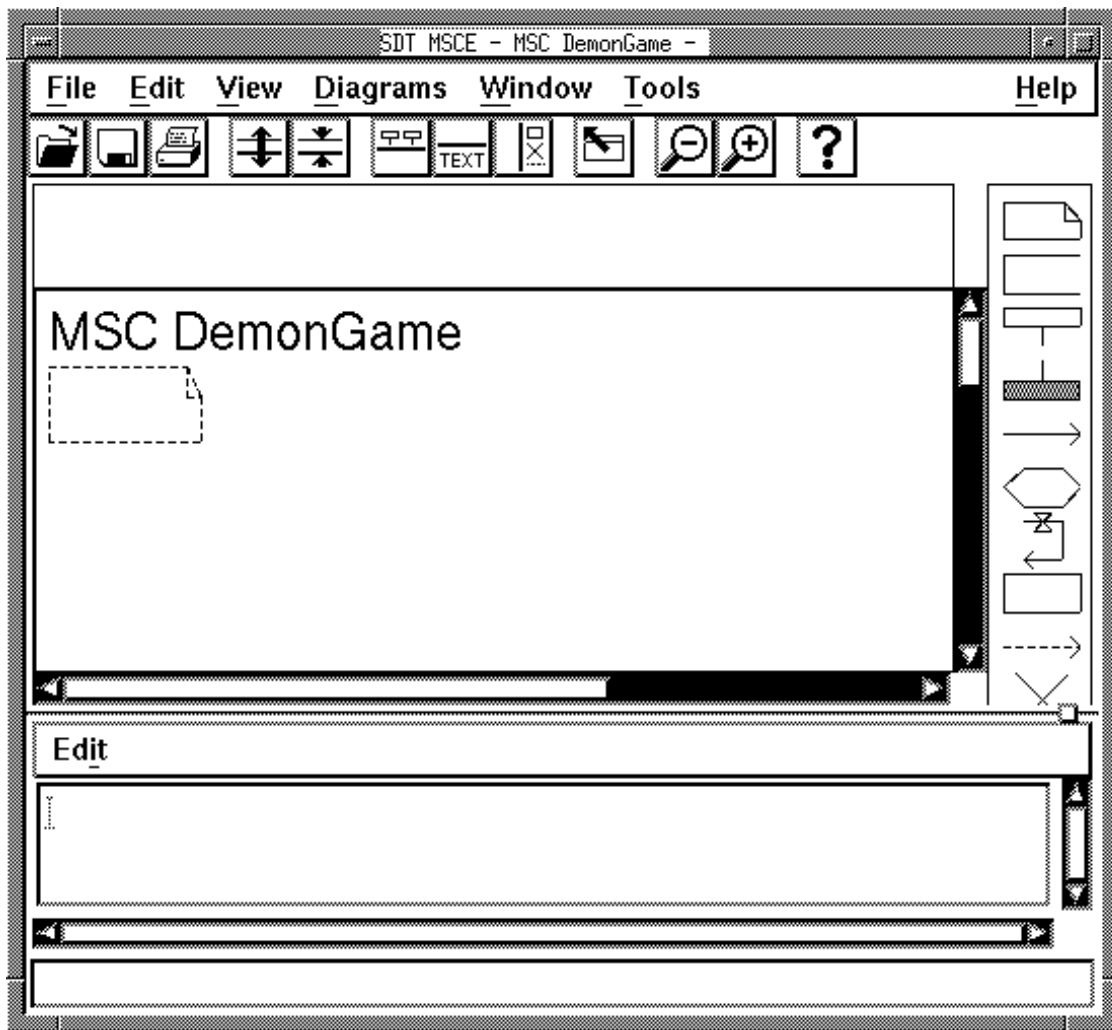


Рис. 23. Окно редактирования диаграмм взаимодействия

В данном уроке мы создадим диаграмму взаимодействия для примера «Игральный Автомат». Окончательный вид диаграммы взаимодействия приведен на Рис. 25.

Диаграмма состоит из четырех объектов (изображаемых вертикальными линиями с прямоугольным заголовком), нескольких символов передачи сообщения (изображаемых горизонтальными линиями, заканчивающихся стрелками), создания процесса (изображаемых пунктирной горизонтальной линией), таймера (изображаемого символом «песочные часы») и двух состояний (изображаемых вытянутыми шестиугольниками). Дополнительно, на диаграмме присутствует текстовый символ, содержащий комментарий.

Для создания диаграммы нужно выполнить следующие операции:

1. Добавьте текстовый символ и введите текст.
2. Добавьте три объекта с именами env, Main и Demon.
 - Выберите символ заголовка объекта в палитре символов, разместите его в области редактирования в соответствии с Рис. 25. Ось объекта создается редактором автоматически.
 - Введите имя объекта (Env, Main, Demon)
 - Введите тип объекта (process Main, process Demon)



Рис. 24. Текстовые атрибуты заголовка объекта

Положение объекта в области редактирования можно изменять, передвигая заголовок объекта.

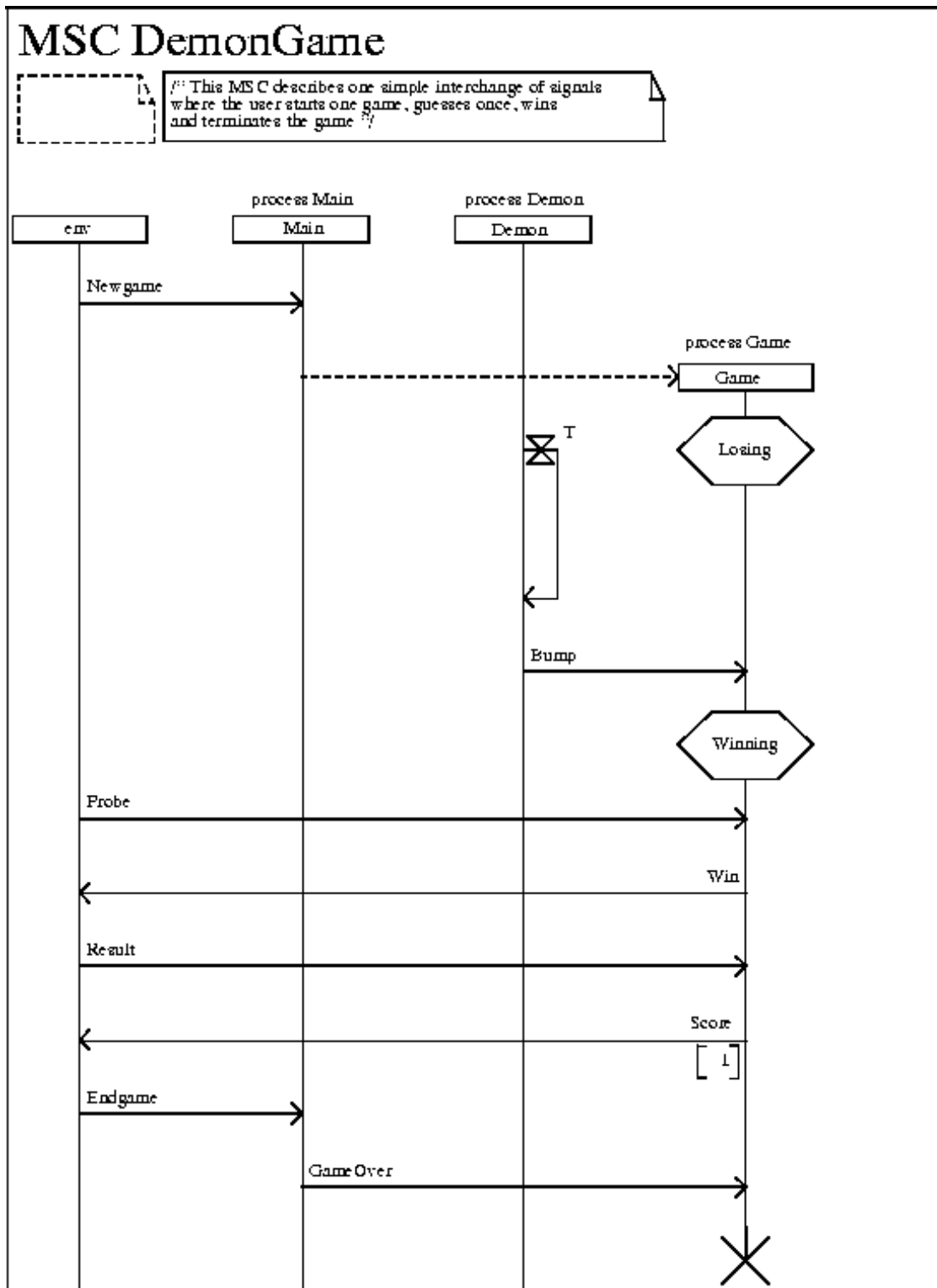


Рис. 25. Диаграмма взаимодействия DemonGame

3. Добавьте сообщение Newgame:
 - Выберите символ сообщения из палитры символов.
 - Укажите точку отправки сообщения на оси объекта env , щелкнув мышью.
 - Укажите точку получения сообщения на оси объекта Main.
 - Введите имя сообщения (Newgame).
 - Положение сообщения можно изменять, перемещая его мышью. Можно отдельно менять точку отправки и точку получения сообщения на осях объектов.
4. Объект Game создается динамически. Добавьте символ порождения процесса (аналогично символу сообщения):
 - Выберите символ порождения процесса из палитры символов.
 - Укажите точку создания нового процесса на оси объекта Main.
 - Разместите заголовок нового объекта, щелкнув мышью в нужном месте области редактирования.
 - Введите тип нового объекта и его имя.
5. Добавьте первый символ состояния на ось объекта Game.
 - Выберите символ состояния в палитре символов и укажите его место на оси объекта, щелкнув мышью. Введите имя состояния: Losing.
 - Символ состояния можно передвигать вдоль оси объекта.
6. Добавьте символ установки таймера на ось объекта Demon.
 - Выберите символ таймера в палитре символов.
 - Укажите точку установки таймера на оси объекта.
 - Укажите точку срабатывания таймера.
 - Введите имя таймера: T.
7. Добавьте все остальные сообщения. Сообщение Score содержит параметр со значением 1. Для ввода значения параметра, выберите нижний прямоугольник.

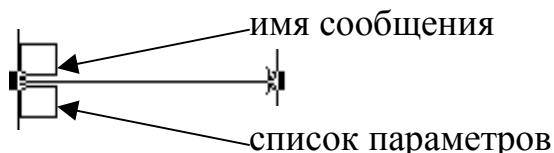


Рис. 26. Текстовые атрибуты сообщения

8. Завершите создание диаграммы, добавив символ остановки процесса:
 - Выберите символ остановки процесса в палитре символов.
 - Поместите его на оси объекта Game.
9. Сохраните диаграмму взаимодействия.

8.3. Заключение

В данной главе были описаны основные приемы работы с редактором диаграмм взаимодействия системы SDT. В результате выполнения упражнений данной главы вы должны были научиться

- создавать диаграммы взаимодействия;
- редактировать диаграммы взаимодействия.

Глава 9. Редактор SDL диаграмм

В данной главе мы создадим SDL-спецификации примера «Игральный автомат».

9.1. Диаграмма системы

Создайте корневой узел дерева структуры SDL системы как показано в упражнениях главы «Организатор системы SDT».

9.1.1. Создание диаграммы системы

Для создания диаграммы системы нужно выполнить следующие операции:

1. Выберите символ системы в дереве системы (в углах символа должны появиться черные квадраты, как показано на Рис. 19)
2. Выберите команду *Edit* в меню *Edit*. Существуют альтернативные возможности перехода к редактированию диаграммы:
 - При нажатии правой кнопки мыши появляется вспомогательное меню, в котором можно выбрать раздел *Edit*, а в нем - команду *Edit*.
 - Самый быстрый переход к редактированию диаграммы – двойной щелчок левой кнопкой мыши на символ диаграммы в дереве SDL системы.
3. Дополнительный диалог *Edit* предложит создать новую диаграмму и открыть окно редактирования (настройка *Show in editor*) (см. Рис. 27).

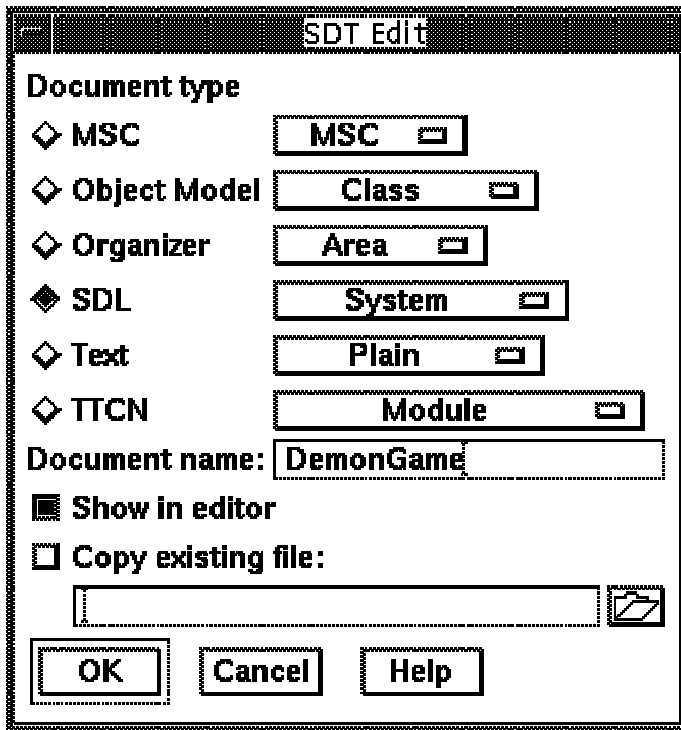


Рис. 27. Диалог создания диаграммы

4. Завершите операцию, нажав на кнопку *OK*.

Верхняя левая часть окна редактирования диаграммы показана на Рис. 28.

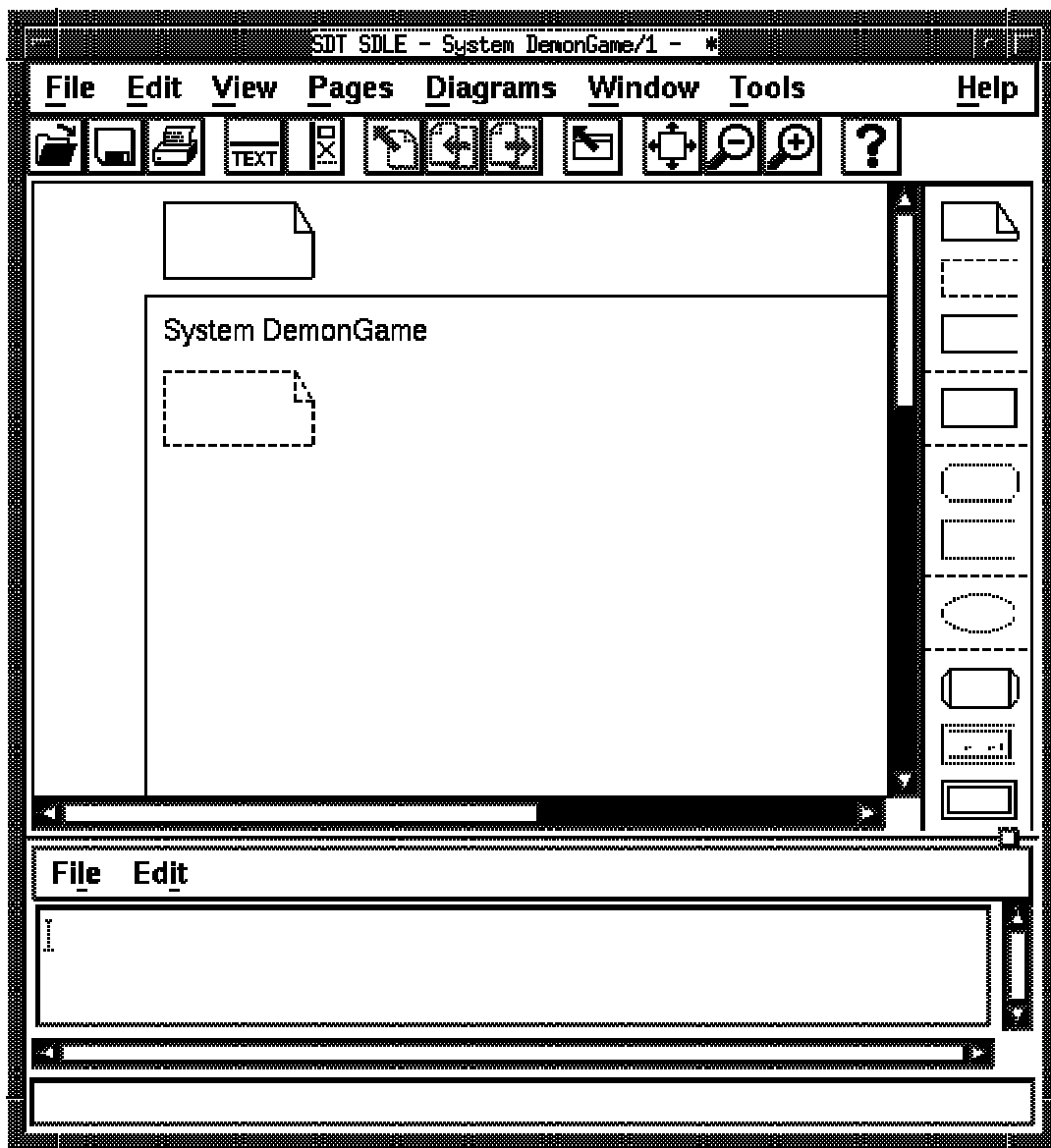


Рис. 28. Окно редактора SDL диаграмм

9.1.2. Редактирование диаграммы системы

Следующим шагом будет заполнение диаграммы. На Рис. 29 показан окончательный вид диаграммы системы DemonGame. Диаграмма состоит из двух блоков: GameBlock и DemonBlock. Блоки соединены каналом с именем С3. Два других канала (С1 и С2) осуществляют взаимодействие системы с окружением. Дополнительно, на диаграмме системы существует текстовый символ, содержащий определения сигналов.

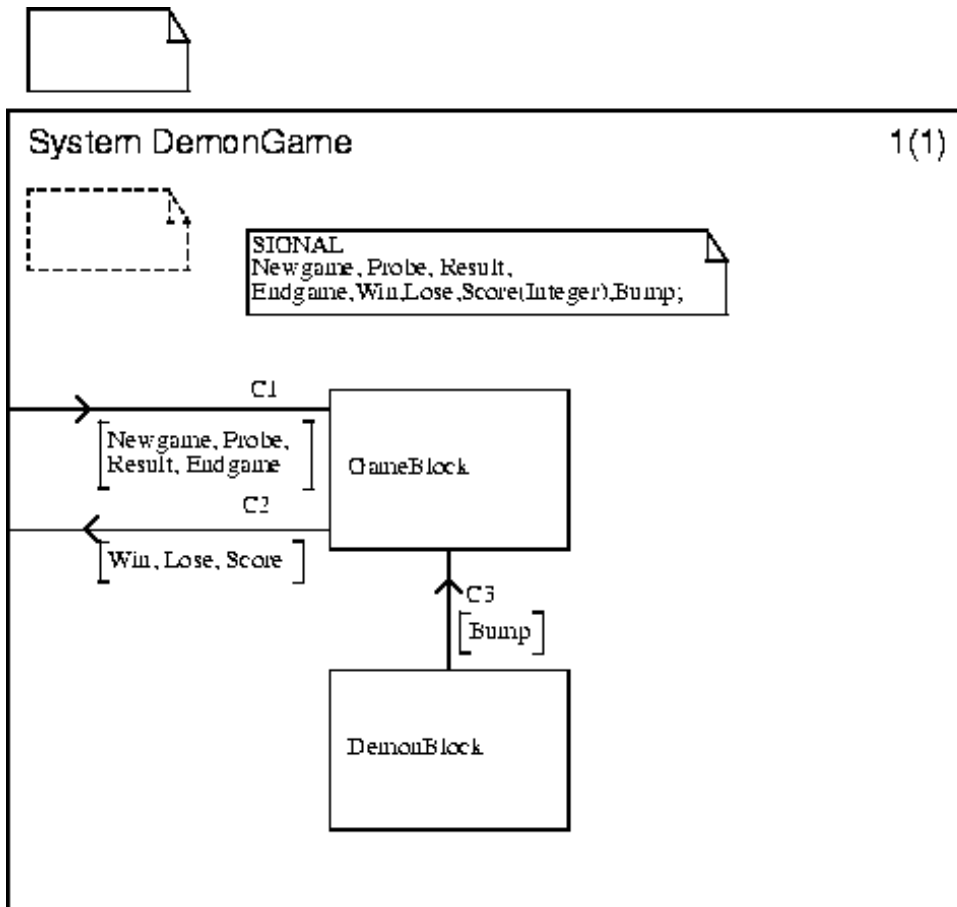


Рис. 29. Диаграмма системы DemonGame

В следующих разделах будет подробно показано, как происходит добавление символов и текста в редакторе SDL диаграмм.

9.1.3. Добавление новых блоков

Для добавления блоков нужно выполнить следующие операции:

1. Выбрать символ блока в палитре символов. Палитра символов расположена вертикально с правой стороны в окне редактора. Символ блока - четвертый сверху.
2. Переместите курсор в область диаграммы. Символ блока «плывет» вслед за курсором. Нажмите на кнопку мыши в том месте, где вы хотите поместить новый блок. Наложение символов недопустимо. В случае если выбранное место для блока приводит к наложению символов, операцию необходимо повторить сначала.

3. После выбора места блока введите имя блока (DemonGame или GameBlock).

- Можно вводить текст внутри символа в текущей позиции курсора. Позицию курсора можно изменить, нажав на кнопку мыши внутри символа.
- Можно вводить текст в текстовом окне (внизу под областью редактирования).

Примечания:

- Редактируемый текст появляется как внутри символа, так и в текстовом окне
- При добавлении нового блока происходит автоматическое обновление дерева структуры системы в окне Организатора.
- Любую операцию можно отменить, выбрав команду *Undo* в меню *Edit*.

9.1.4. Перемещение символов и изменение их размеров

Для перемещения символа на диаграмме нужно нажать левую кнопку мыши внутри данного символа и, не отпуская кнопки, переместить символ на новое место.

Для изменения размеров символов нужно потянуть за один из уголков символа. Курсор должен быть достаточно близко к уголку. Если не получается - нужно сначала выбрать данный символ, нажав кнопку мыши внутри символа, а затем повторить операцию.

9.1.5. Создание каналов между блоками

Для создания канала от блока DemonBlock к блоку GameBlock:

1. Выбрать блок DemonBlock. В нижней части символа появится "ушко" (handle).

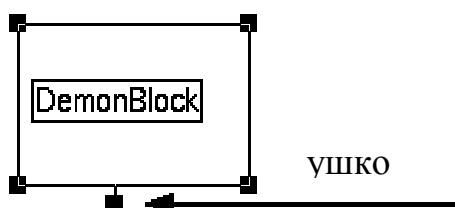


Рис. 30. Ушко

2. «Потяните» за «ушко». Как только редактор зафиксирует движение «ушка», начнется отрисовка линии канала. С этого момента кнопку мыши можно отпустить.

3. Щелкните мышью внутри блока GameBlock. Произойдет подсоединение канала.

С каждым каналом связаны два текстовых атрибута для ввода имени канала и списка сигналов, допустимых для передачи по данному каналу.

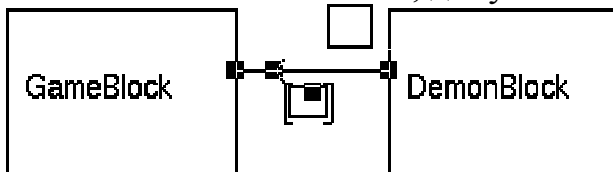


Рис. 31. Текстовые атрибуты канала

Сразу после создания канала введите его имя (С3). Иногда нужно предварительно выбрать символ канала.

Для задания списка сигналов, передаваемых по каналу (сигнал Bump) нужно проделать следующие операции:

1. Выберите текстовое поле, окруженное квадратными скобками [и]. (Для этого надо щелкнуть мышью в пространстве между квадратными скобками).
2. Введите имя сигнала.
3. Текстовые атрибуты канала можно перемещать по диаграмме.

9.1.6. Создание каналов к окружению

Для создания канала от блока к окружению системы (например, С2):

1. Выберите блок.
2. Потяните за «ушко» и завершите операцию, нажав кнопкой мыши на рамку диаграммы (см. рис.12).
3. Введите имя канала и сигналы.

Для создания канала от окружения к блоку (например, С1):

1. Создайте канал от блока к окружению (см. выше).
2. Выберите команду *Redirect* из меню *Edit*.

9.1.7. Создание текстового символа

Для создания текстового символа с определениями сигналов нужно проделать следующие операции:

1. Выберите текстовый символ в палитре символов, переместите его в область редактирования и введите определения сигналов, как показано на Рис. 29.

9.1.8. Сохранение диаграммы

1. Для сохранения диаграммы нужно выбрать команду *Save* из меню *File* редактора SDL диаграмм.

Переход в окно редактора SDL диаграмм происходит по двойному щелчку мышью по диаграмме системы.

2. Выполнение команды сохранения приводит к появлению стандартного диалога выбора файла. Этот диалог используется системой SDT всякий раз, когда требуется выбрать файл (при открытии, сохранении и т.д.). Заголовок диалога соответствует выполняемой операции (в данном случае - *SDT Save*) (см. Рис. 32).

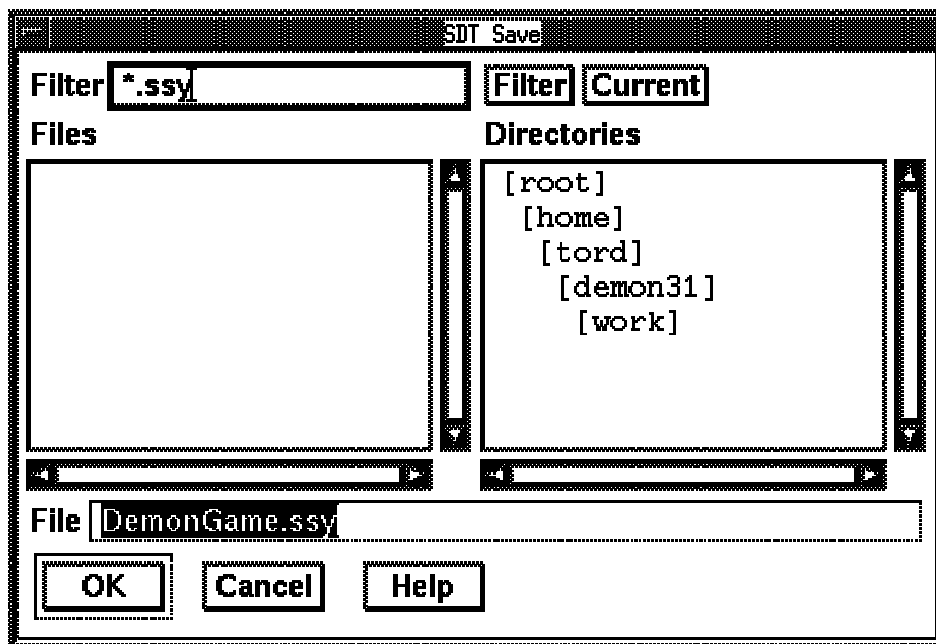


Рис. 32. Стандартный диалог выбора файла

3. Редактор SDL диаграмм предлагает сохранить диаграмму системы в файле *DemonGame.ssy*. Суффикс **.ssy* является стандартным для диаграмм систем. Завершите операцию, нажав кнопку *OK*. Диаграмма сохранена в файле. Имя файла отображается в заголовке окна редактора. Проверьте, что в окне Организатора имя файла диаграммы системы изменилось с *[unconnected]* на *DemonGame.ssy*. Два других символа в дереве системы по-прежнему остаются *[unconnected]*. Они представляют собой символы блоков на диаграмме системы.

9.2. Проверка синтаксиса диаграммы системы

После выполнения упражнений данного урока вы должны научиться:

- Запускать Анализатор;
- Устанавливать настройки Анализатора;
- Работать с окном сообщений Организатора;
- Находить и исправлять синтаксические ошибки;

9.2.1. Запуск Анализатора

Для проверки синтаксической правильности созданной диаграммы нужно использовать Анализатор. Анализатор полностью интегрирован в системы SDT. Анализатор запускается из Организатора следующим образом.

1. Выберите диаграмму системы в Организаторе. Затем, выберите команду *Analyze* из меню *Generate* Организатора.
2. В случае если имеются измененные, но не сохраненные диаграммы, перед запуском Анализатора будет предложено сохранить измененные диаграммы. Диалог *Save* появляется в этом случае автоматически (см. рис. 16). Для сохранения всех измененных диаграмм, нажмите кнопку *Save All*.
3. После завершения диалога *Save* появляется диалог настройки Анализатора (*Analyzer options*) (см. Рис. 33).

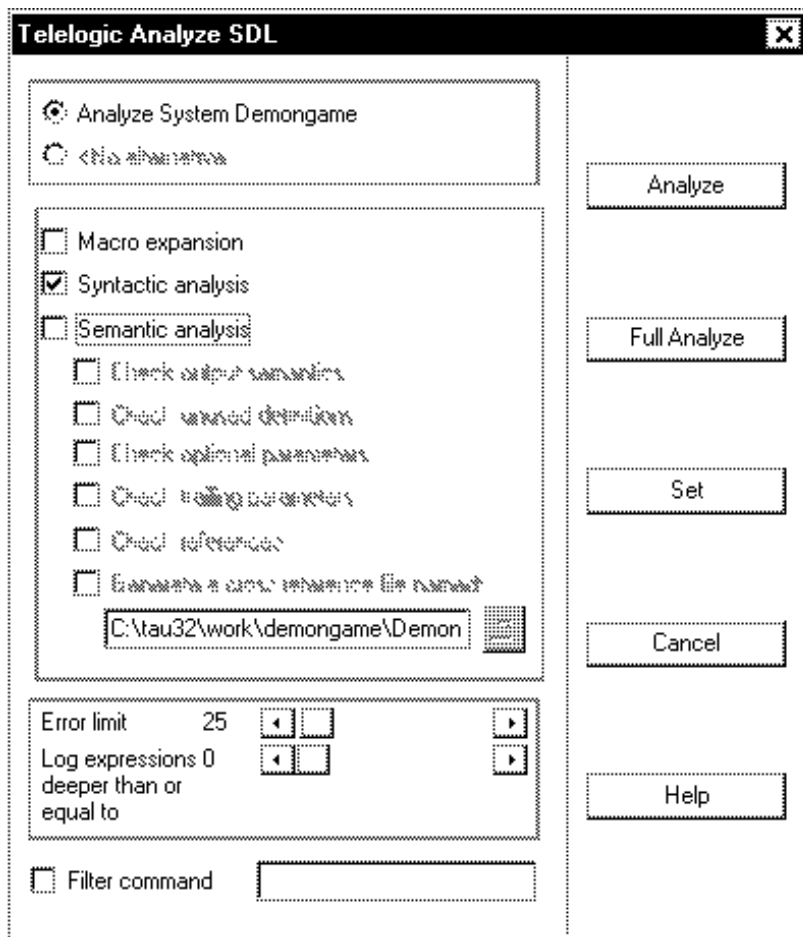


Рис. 33. Диалог настройки Анализатора

4. Для проверки синтаксиса диаграммы выберите установки как показано на Рис. 33:

- *Macro expansion* выключено
- *Syntactic analysis* включено
- *Semantic analysis* выключено
- Выберите подходящее значение предельного числа сообщений об ошибках (*error limit*). Данная установка определяет, сколько сообщений об ошибках и предупреждений может быть выдано Анализатором перед тем, как закончить обработку некорректной программы.

5. Нажмите кнопку *Analyze*. Анализатор начинает обработку диаграммы. После окончания работы, Анализатор выведет сообщение "Analyzer done" в область статуса.

6. Для выдачи детальных сообщений в Организаторе предусмотрено специальное текстовое окно *Organizer Log*. Обычно окно сообщений появляется автоматически каждый раз, когда обнаружена информация типа

«предупреждение» или «сообщение об ошибке». Можно открыть окно сообщений вручную, выбрав команду *Organizer Log* в меню *Tools* Организатора, или нажав на специальную «горячую клавишу».

9.2.2. Где искать сообщения о синтаксических ошибках

Все сообщения Анализатора попадают в специальное *окно сообщений (Organizer Log)*. В конце списка сообщений выдается статус следующего вида:

```
-----  
Number of errors:      xxxx  
+ Analysis completed
```

Если диаграмма синтаксически корректна, строка с указанием числа ошибок будет отсутствовать.

1. Попробуйте внести синтаксические ошибки в диаграмму системы. Например, удалите одну из запятых в списке сигналов канала C1.
2. Сохраните диаграмму и повторите проверку синтаксиса.

9.2.3. Как интерпретировать синтаксические ошибки

При обнаружении синтаксической ошибки в списке сообщений появляется текст вида:

```
#SDTREF(SDL, /home/nick/course/demongame(1), 131(25, 50), 3  
, 8)  
ERROR 312 Syntax error in rule SIGNALLIST, symbol  
Name found but one of the following expected:  
, ; comment  
Result Endgame;  
?
```

Сообщение об ошибке имеет следующий смысл:

1. Первая часть сообщения (`#SDTREF ...`) представляет собой графическую ссылку на исходную диаграмму, указывающую файл, страницу, символ, номер строки и номер позиции в строке, где была обнаружена ошибка.
2. Вторая часть сообщения (`ERROR 312 ...`) содержит номер сообщения и пояснительный текст.

3. Заключительная часть сообщения (Result Endgame) и символ ? воспроизводят фрагмент, в котором произошла ошибка.

Для исправления ошибки можно автоматически выбрать нужную диаграмму и символ. Для этого нужно выполнить следующие действия:

1. Выделите мышью текст сообщения об ошибке.
2. Выберите команду *Show Error* в меню *Tools* Организатора.
3. В результате выполнения команды редактор SDL диаграмм показывает нужную диаграмму и делает текущим символ, содержащий ошибку. Исправьте ошибку.
4. Для исправления следующей ошибки можно нажать еще раз «горячую клавишу» *Show Error*. Система SDT автоматически покажет место следующей ошибки (если таковые имеются).

Окно сообщений можно очистить, выбрав команду *Clear Log* из меню *Edit*,

9.3. Диаграммы блоков

9.3.1. Создание диаграммы блока из Организатора

После выполнения упражнений данного урока вы должны научиться:

- Создавать диаграмму блока
- Работать с несколькими диаграммами в редакторе

Для создания диаграммы блока из Организатора нужно выполнить следующие операции:

1. Перейти в окно Организатора и сделайте двойной щелчок по символу *GameBlock*. Это приведет к появлению уже знакомого диалога *Edit* (см. рис. 7). Убедитесь в том, что настройка *Show in editor* включена. Нажмите кнопку *OK*. Это приведет к появлению диалога добавления страницы *Add Page* (Рис. 34).

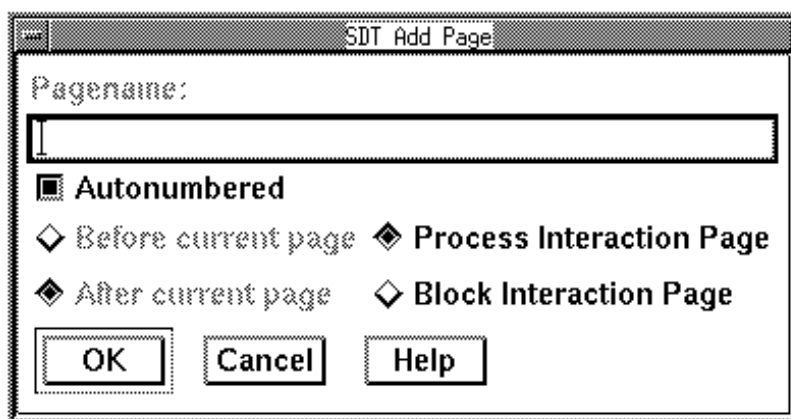


Рис. 34. Диалог добавления страницы

2. Диалог добавления страницы служит для указания вида диаграммы блока: блок, содержащий процессы или блок, содержащий другие блоки. В диалоге можно явно ввести имя новой страницы. По умолчанию, система SDT предлагает автоматическую нумерацию страниц (*Autonumbered*). Выберите вид страницы *Process Interaction Page* (блок, содержащий процессы) и нажмите кнопку *OK*.

3. Редактор SDL диаграмм открывает вновь созданную диаграмму блока GameBlock на странице 1. Палитра символов содержит другие элементы по сравнению с окном редактирования диаграммы системы.

На Рис. 35 показан окончательный вид блока GameBlock. Диаграмма содержит два символа процесса, пять каналов, три соединения и текстовый символ, в котором находится определение сигнала.

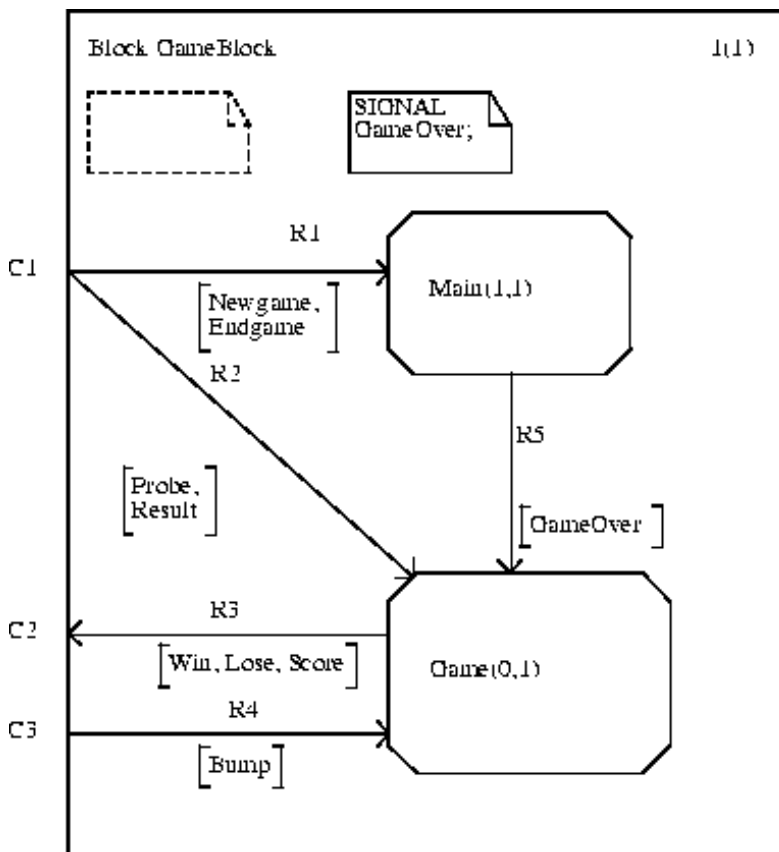


Рис. 35. Диаграмма блока GameBlock

Вам предстоит создать диаграмму GameBlock в соответствии с Рис. 35. Ниже представлена дополнительная информация о редактировании новых символов.

9.3.1.1.1 Имя процесса и число экземпляров процесса

При создании символа процесса кроме имени процесса нужно дополнительно ввести число экземпляров процесса. Число экземпляров процесса вводится непосредственно после имени процесса и заключается в круглые скобки.

9.3.1.1.2 Межпроцессные каналы

После выделения символа процесса появляются два «ушка».

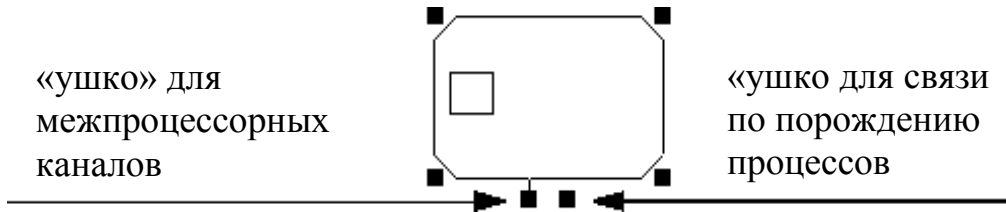


Рис. 36. Два «ушка» в символе процесса

- Левое «ушко» используется для создания каналов между процессами.
- Правое «ушко» используется для задания связи по порождению ("отец-сын").

Соединения

При соединении канала к символу рамки, помимо имени канала и списка сигналов, необходимо задать соединение межпроцессного канала с межблочным каналом на родительской диаграмме (диаграмме системы). Для задания имени межблочного канала используется дополнительное текстовое поле с внешней стороны рамки (см. Рис. 37).

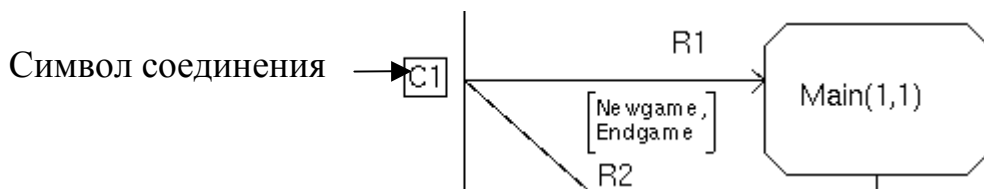


Рис. 37. Символ соединения

Для редактирования соединения нажмите кнопку мыши внутри нужного текстового поля и введите имя канала.

9.3.2. Создание диаграммы блока по существующей копии

В результате упражнений данного урока вы должны научиться следующему:

- Создавать диаграмму по существующей копии
- Сохранять диаграмму в новом файле

Диаграмму блока GameBlock вы создали, сделав двойной щелчок мышью по символу блока в Организаторе. Заметим, что команда *Edit* может быть выполнена также при двойном щелчке на символ блока в редакторе SDL диаграмм (т.е. на диаграмме системы).

Для создания блока DemonBlock выполните следующую последовательность операций:

1. Выберите символ блока DemonBlock в Организаторе или в редакторе. Сделайте двойной щелчок мышью по этому символу. Появится диалог редактирования *Edit*:

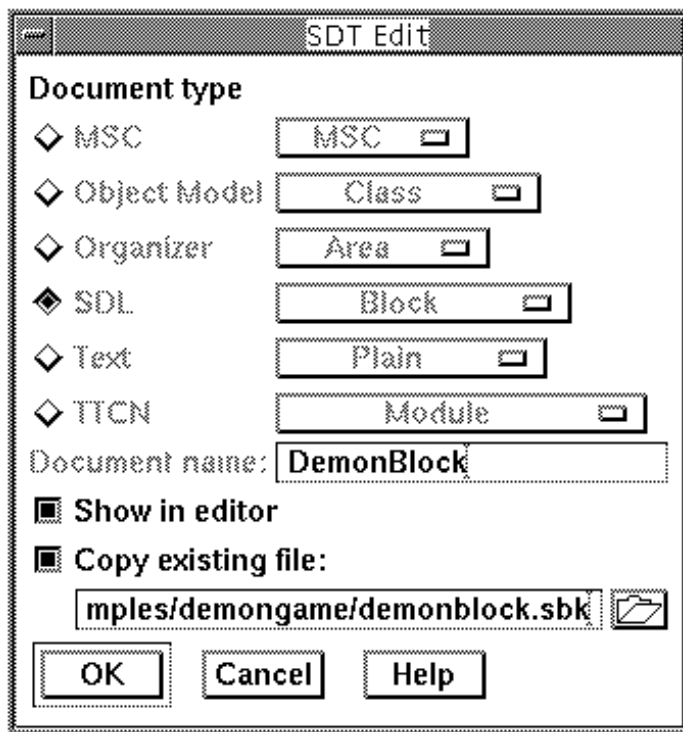


Рис. 38. Диалог создания блока DemonBlock

Создадим блок путем копирования уже существующего файла. Имя директории `$examples/demongame`. Имя файла, содержащего блок DemonBlock `demonblock.sbk`

2. Убедитесь в том, что установка *Copy existing file* включена.

3. Укажите файл, содержащий блок DemonBlock одним из следующих способов:

- введите полный путь к файлу (включая директорию); либо
 - Нажмите на кнопку *folder* справа от текстового поля ввода. Появится стандартный диалог выбора файла с названием *Select File to Copy Block DemonBlock*.
 - Для перемещения по директориям нужно сделать двойной щелчок мышью по директории. Файл `demonblock.sbk` должен появиться на левой панели. Выберите этот файл и завершите операцию, нажав кнопку *OK*.
4. Завершите диалог *Edit*, нажав на кнопку *OK*. Редактор откроет диаграмму в текущем окне редактирования.

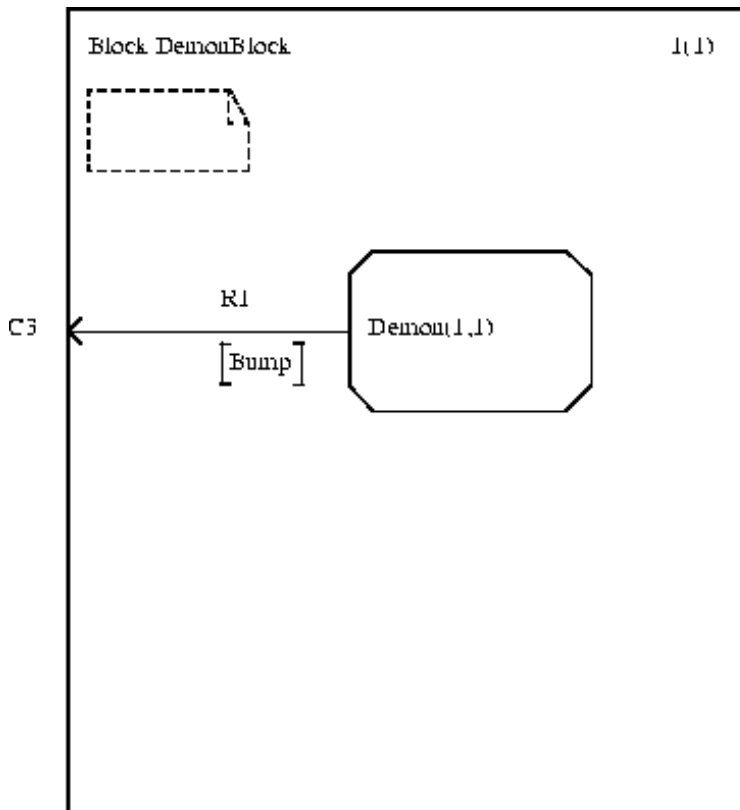


Рис. 39. Диаграмма блока DemonBlock

5. Теперь, сохраните диаграмму (нажмите «быструю клавишу» *Save*). Появится диалог выбора файла. Система SDT предложит имя файла `DemonBlock.sbk`.
6. Завершите операцию, нажав кнопку *OK*.

9.4. Работа с несколькими диаграммами

Вы создали две SDL диаграммы. Обе диаграммы открыты в редакторе SDL диаграмм, хотя показывается только одна из них.

В редакторе имеется меню *Diagrams*, в котором находится список всех открытых диаграмм и отдельных страниц.

1. Выберите меню *Diagrams*. Меню должно содержать две строки:



Рис. 40. Меню открытых диаграмм

Каждая строка в меню соответствует одной диаграмме и странице, открытой в редакторе. Справа показано имя файла, в котором сохранена диаграмма.

9.5. Диаграммы процессов

9.5.1. Создание диаграммы процесса

Вы создали все структурные элементы SDL системы. Теперь нужно описать поведение системы, т.е. создать графы процессов.

9.5.2. Редактирование диаграммы процесса *Demon*

В результате выполнения упражнений данного урока вы должны научиться:

- Добавлять символы по двойному щелчку мыши;
- Работать с буфером копирования;
- Вставлять символы в участок;
- Получать информацию о грамматике языка SDL;

Диаграмма процесса *Demon* приведена на Рис. 41.

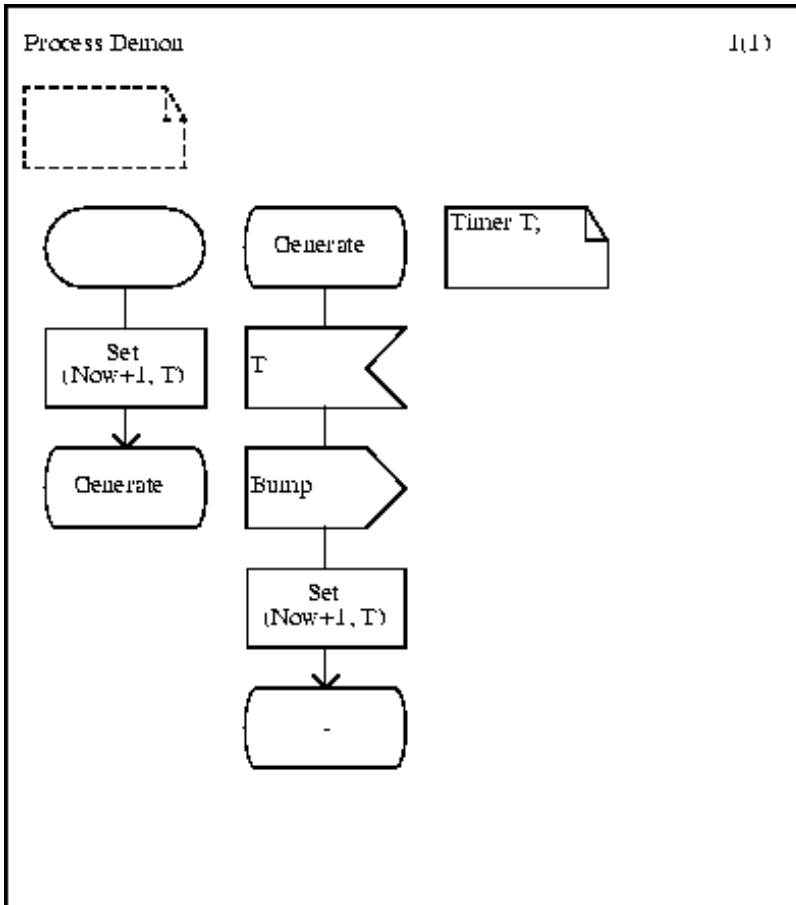


Рис. 41. Диаграмма процесса Demon

1. Когда появится диалог добавления страницы, задайте тип новой страницы *Graph Page*.

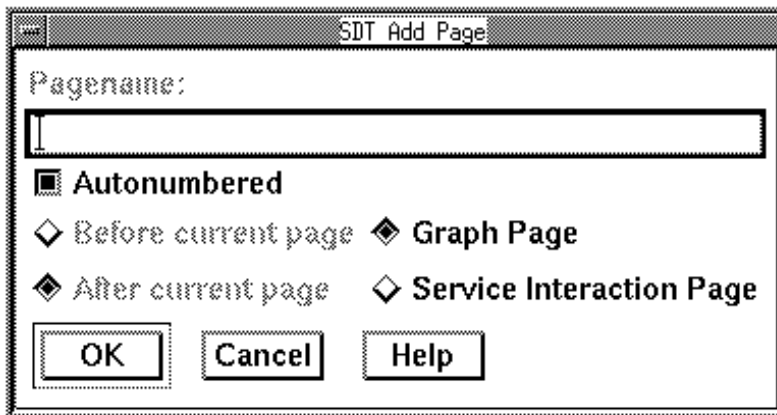


Рис. 42. Установка типа страницы для диаграммы процесса

2. Редактор SDL диаграмм создает диаграмму процесса. Обратите внимание, что палитра символов выглядит иначе, чем при редактировании диаграммы блока.

Как показано на Рис. 41, диаграмма состоит из двух участков. Редактор может автоматически соединять символы линиями при добавлении новых символов к участку. Каждый символ на диаграмме содержит определенный текст. Текст можно вводить в любое время, например, сразу после создания нового символа или после того, как созданы все символы.

9.5.2.1.1 Создание левого участка

Для создания левого участка нужно выполнить следующие операции:

1. Выбрать стартовый символ из палитры символов и поместить его в область редактирования.
2. Сделайте двойной щелчок мышью по символу действия. Пустой символ действия будет добавлен к стартовому символу.
3. Для ввода теста в символ действия воспользуемся контекстной информацией по грамматике языка SDL.

Выберите команду *Grammar Help* из меню *Windows*. Появится окно информации по грамматике SDL.

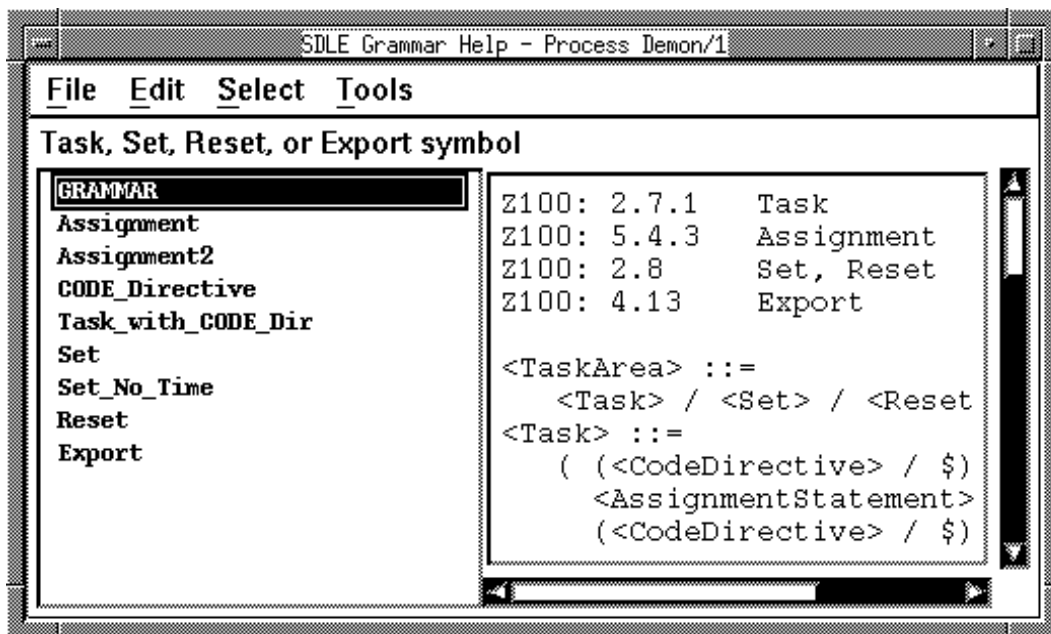


Рис. 43. Окно информации по грамматике SDL

Список на левой стороне окна описывает несколько «ситуаций» использования текущего символа. Верхний элемент списка – GRAMMAR – описывает полную грамматику для текущего символа.

В верхней части на правой стороне окна содержатся ссылки на стандарт языка SDL. Далее на этой панели приводятся предложения БНФ для всех конструкций, допустимых для данного символа.

4. Выберите «ситуацию» установки таймера (set).

Теперь на правой стороне окна показана БНФ для установки таймера: SET (Now+Expr, TimerName) .

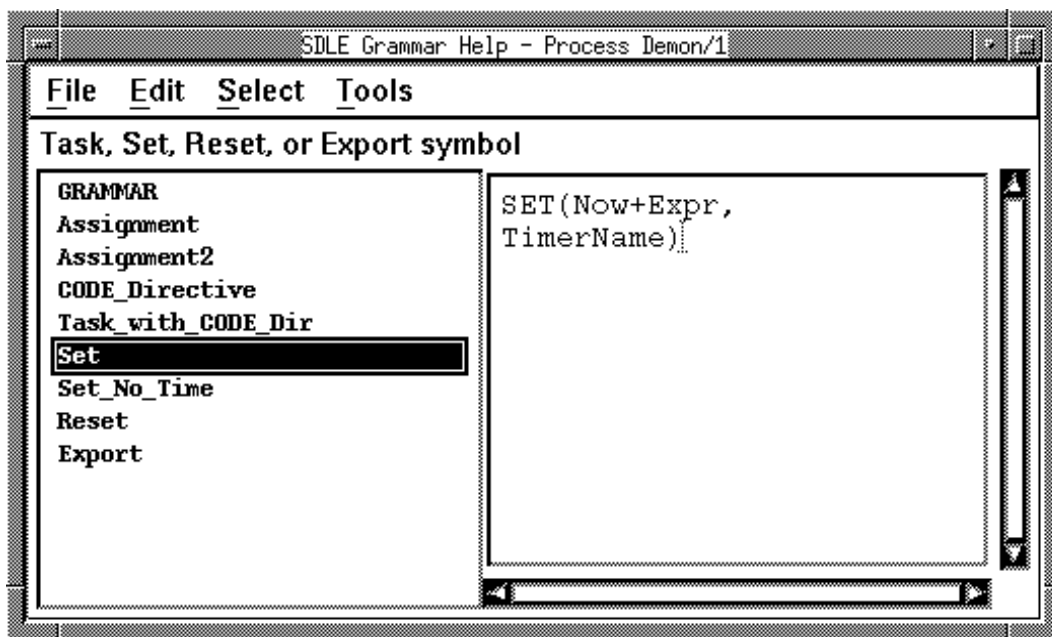


Рис. 44. Грамматика для установки таймера

5. Для ввода конструкции в текущий символ выберите команду *Insert* из меню *Edit*. Более быстрый способ ввода текста в символ – двойной щелчок мышью по названию «ситуации» в списке на левой стороне окна.
6. Замените имена нетерминалов Expr и TimerName на конкретные выражения (1 и T, соответственно). Для редактирования участка текста можно использовать окно редактирования текста. Выделите мышью участок, подлежащий изменению, и введите новый текст.

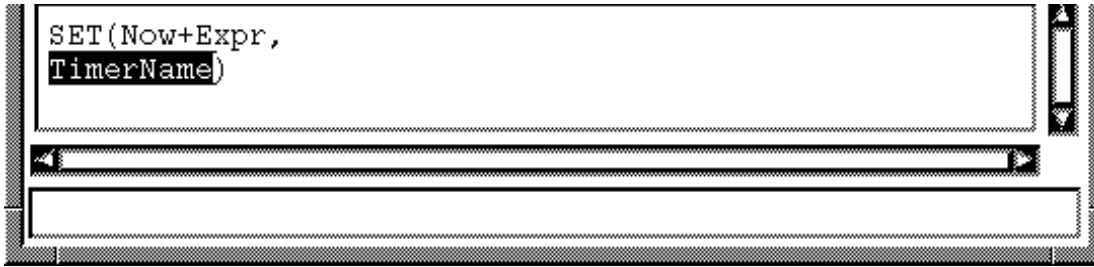


Рис. 45. Окно редактирования текста

Для закрытия окна информации по грамматике выберите команду *Close* в меню *File*.

7. Завершите создание участка, сделав двойной щелчок по символу состояния. Введите текст: *Generate*.

Создание правого участка

Для создания правого участка нужно выполнить следующие операции:

1. Скопируйте символ состояния в буфер. Команды работы с буфером находятся в меню *Edit* или в отдельном меню, всплывающем при нажатии правой кнопки мыши.
2. Вставьте копию символа из буфера, выполнив команду *Paste*. Поместите новый символ и завершите операцию, нажав левую кнопку мыши.
3. Добавьте символ ввода сигнала двойным щелчком мыши. Введите текст: *T*.
4. Добавьте символ отправки сигнала двойным щелчком мыши. Введите текст: *Bump*.
5. Скопируйте символ действия с текстом *SET (NOW + 1, T)*.
6. Сделайте текущим символ отправки сигнала *Bump*. Нажмите правую кнопку мыши и выберите команду *Insert Paste*. Данная команда одновременно копирует символ из буфера и добавляет его к участку.
7. Завершите создание участка, сделав двойной щелчок мышью по символу состояния. Введите текст: *-*.
8. Добавьте текстовый символ и введите определение таймера *T*.
9. Сохраните диаграмму.

9.5.3. Редактирование процесса *Game*

Упражнения данного урока помогут вам научиться:

- Редактировать параллельные участки
- Соединять символы

Создайте диаграмму процесса Game. Окончательный вид диаграммы процесса Game приведен на рисунке 34.

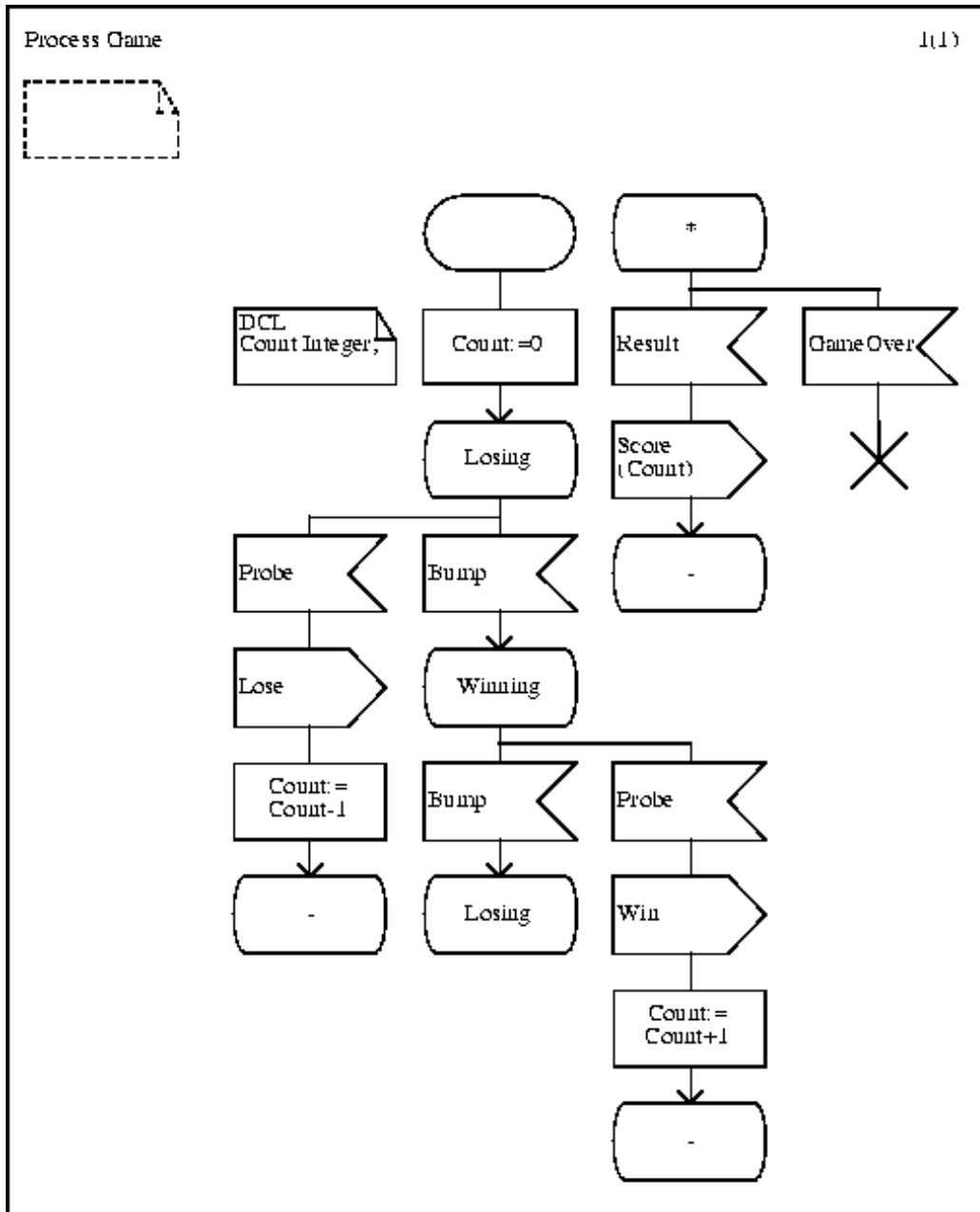


Рис. 46. Диаграмма процесса Game

Редактирование стартового перехода

1. Создайте стартовый символ, символ действия и символ состояния Losing.
2. Теперь будем добавлять два параллельных символа ввода сигналов. Убедитесь в том, что символ состояния является текущим. Нажмите

клавишу <Shift> и, не отпуская ее, сделайте два двойных щелчка мышью по символу ввода сигнала в палитре символов.

- Отпустите клавишу <Shift> и сделайте текущим левый символ ввода сигнала.
- Введите имя сигнала Probe и завершите левый участок.
- Выберите правый символ ввода сигнала. Введите имя сигнала Bump и завершите прямую ветвь правого участка (без дополнительной ветви, связанной с вводом сигнала Probe в состоянии Winning).
- Выберите символ ввода сигнала Probe на левом участке. Выберите команду *Select Tail* из меню *Edit*. Данная команда отмечает все символы до конца текущего участка.
- Сделайте копию отмеченного участка и добавьте ее в диаграмму.
- Измените имя сигнала в символе ввода с Lose на Win.
- Измените текст в символе действия на `Count := Count+1`
- Соедините символ состояния Winning с символом ввода сигнала Probe. Для этого выберите символ состояния, «потяните» за «ушко» и, не отпуская кнопки мыши, подведите линию к символу ввода сигнала.

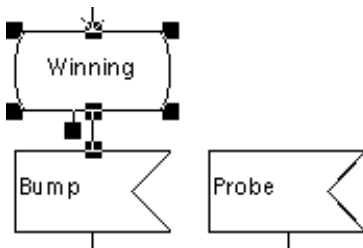


Рис. 47. Символ состояния и его «ушко»

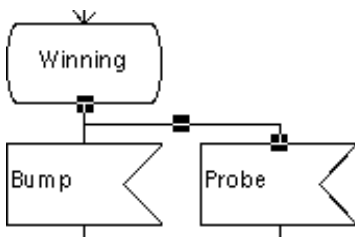


Рис. 48. Состояние после присоединения новой ветви

- Завершите редактирование диаграммы в соответствии с Рис. 49.

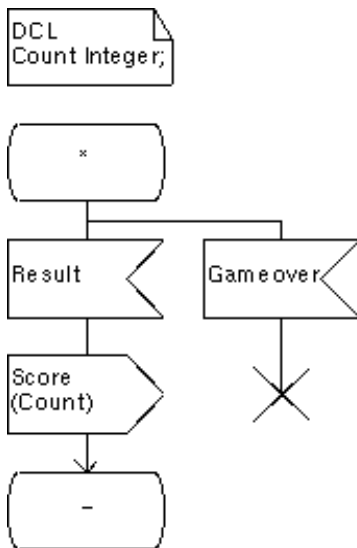


Рис. 49. Оставшиеся части диаграммы

9.5.4. Редактирование процесса Main

Окончательный вид диаграммы процесса Main приведен на Рис. 50.

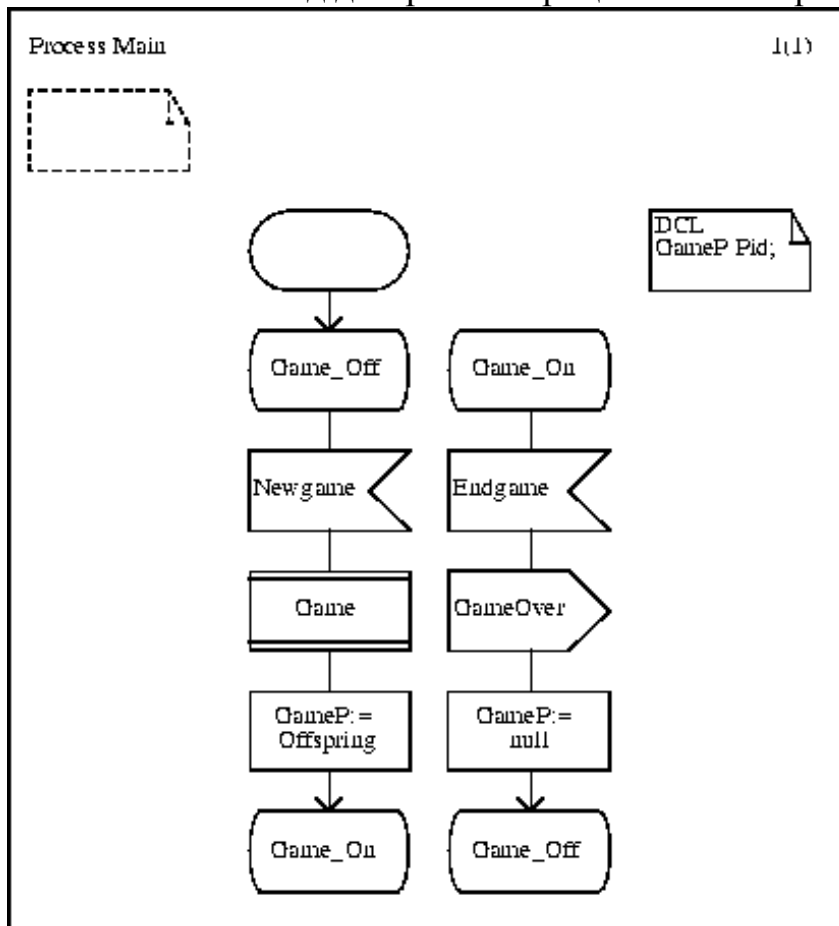


Рис. 50. Диаграмма процесса Main

9.6. Полный анализ системы

Упражнения данного урока помогут вам научиться:

- запускать синтаксический и семантический анализ всей системы

Для выполнения полного анализа системы нужно выполнить следующие операции:

1. Выбрать символ системы в окне Организатора.
2. Выбрать команду *Analyze* в меню *Generate*.
3. Изменить настройки Анализатора, как показано на Рис. 51.

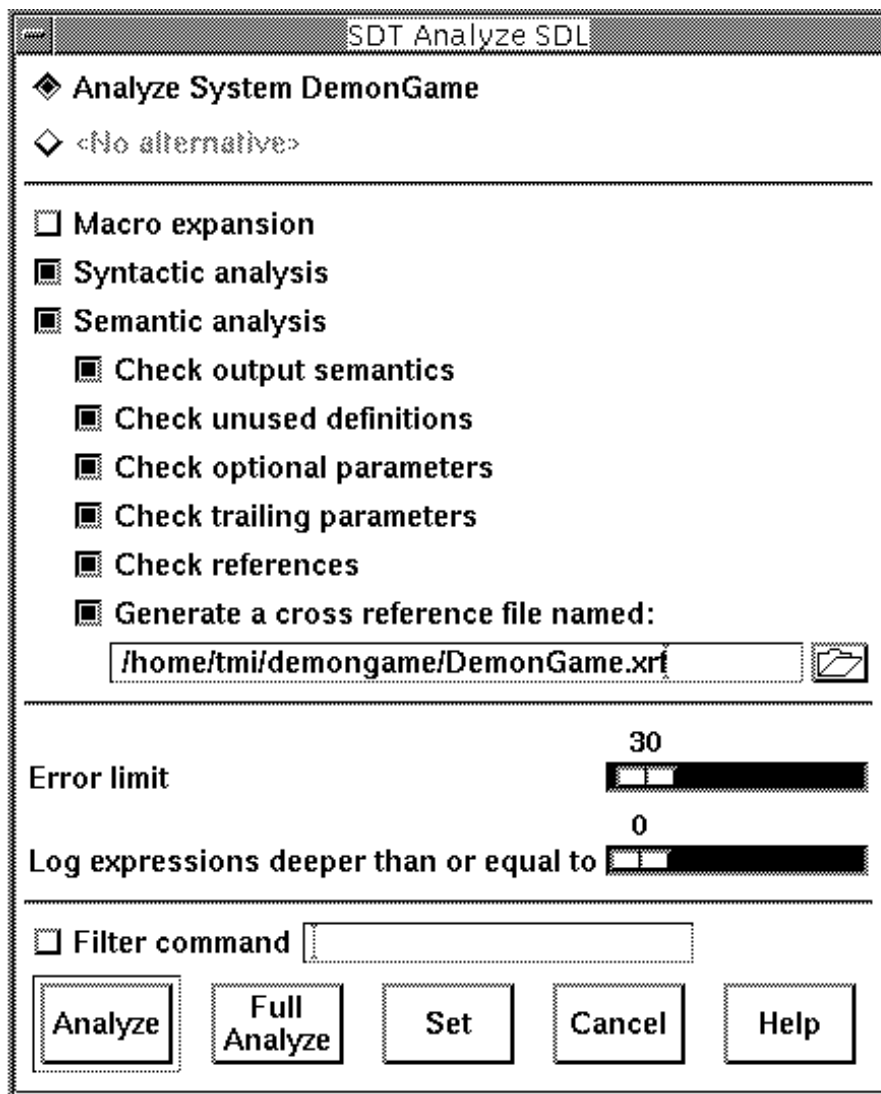


Рис. 51. Настройки Анализатора

4. Для запуска Анализатора нажмите на кнопку *Analyze*.

5. После завершения работы Анализатора в окне статуса появится сообщение "Analyzer done". Убедитесь в том, что трасса Анализатора не содержит сообщений об ошибках. В конце трассы должен быть следующий текст:

```
+ Analysis started
Conversion of SDL to PR started
Conversion to PR completed
Syntactic analysis started
Syntactic analysis completed
Semantic analysis started
Semantic analysis completed
+ Analysis completed
```

9.7. Заключение

В данной главе были описаны основные приемы работы с редактором SDL диаграмм системы SDT. В результате выполнения упражнений данной главы вы должны были научиться следующему:

- Запускать Анализатор;
- Устанавливать настройки Анализатора;
- Работать с окном сообщений Организатора;
- Находить и исправлять синтаксические ошибки;
- Создавать диаграмму блока
- Работать с несколькими диаграммами в редакторе
- Создавать диаграмму по существующей копии
- Сохранять диаграмму в новом файле

Глава 10. Выполнение SDL спецификаций

После того как созданы и проанализированы все диаграммы системы, можно автоматически сгенерировать исполняемую версию системы и осуществить ее запуск в интерактивном режиме, т.е. непосредственно проанализировать поведение системы.

После выполнения упражнений данного урока вы должны научиться:

- создавать исполняемую систему;
- запускать Монитор исполняемой системы;
- запускать систему из графического интерфейса Монитора;

10.1. Создание исполняемой системы

Для создания исполняемой системы нужно выполнить следующие операции:

1. Выбрать символ системы в окне Организатора
2. Выбрать команду *Make* из меню *Generate*. Выполнение команды приведет к появлению диалога настройки функции *Make*.

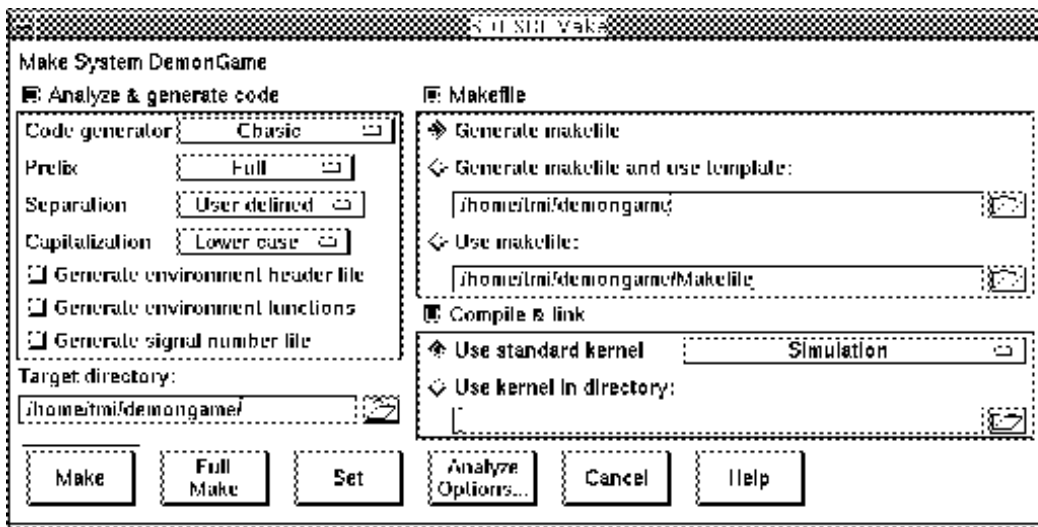


Рис. 52. Диалог настройки функции *Make*

3. Установите настройки, как показано на Рис. 52.
 - *Analyze & generate code* включено.
 - *Compile & link* включено
 - *Generate makefile* включено
 - *Use standard kernel* включено. Убедитесь, что выпадающее меню *Use standard kernel* установлено в *Simulation* (стандартное ядро для выполнения спецификаций).

4. Завершите выполнение команды, нажав на кнопку *Make*.
5. Проверьте, что при анализе системы не выдано сообщений об ошибках. В окне статуса Организатора должно появиться сообщение "Compiler done", а в окне *Organizer Log* нет никаких сообщений об ошибках между строками «Make started» и «Make completed»
6. В случае, если были выданы сообщения об ошибках, вызовите снова диалог настройки функции *Make*, но нажмите на этот раз кнопку *Full Make*. Теперь никаких сообщений об ошибках быть не должно.

10.2. Запуск спецификации на выполнение

После выполнения команды *Make* исполняемая система находится в файле с именем `demongame_xxx.sct` в той директории, откуда была запущена система SDT (суффикс `_xxx` определяется конкретной платформой и используемым компилятором). Исполняемая система содержит *Монитор*, который предоставляет набор команд для наблюдения за выполнением системы и управлением хода выполнения.

Исполняемую систему можно запустить непосредственно из командной строки. В этом случае команды Монитора нужно будет вводить непосредственно в текстовой форме.

В системе SDT имеется также графический интерфейс к Монитору. Запуск графического интерфейса Монитора осуществляется из Организатора следующим образом.

1. Выберите команду *Simulator UI* из дополнительного меню *SDL* в меню *Tools*.
2. Через несколько секунд появится окно графического интерфейса Монитора.

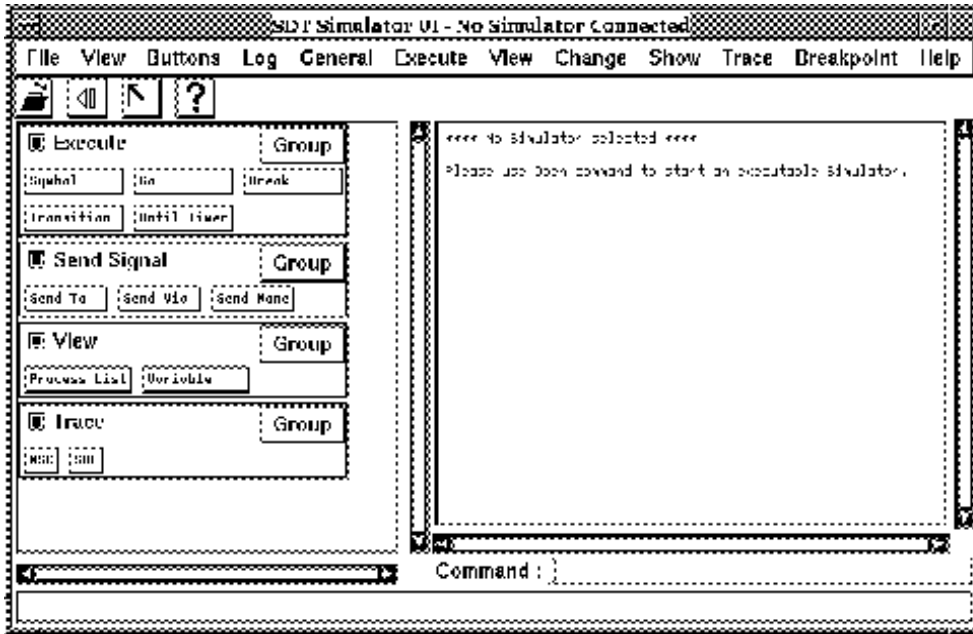


Рис. 53. Окно графического интерфейса Монитора

Текстовое окно (правая панель) содержит сообщения Монитора такие, как введенные команды, результаты команд, сообщения об ошибках. Начальное сообщение указывает, что исполняемая система не запущена.

3. Для запуска исполняемой системы выберите команду *Open* из меню *File* или нажмите «быструю клавишу» *Open*.
4. В диалоге выбора файла нужно указать файл `demongame_xxx.sct`. Завершите выполнение команды, нажав на кнопку *OK* или *Open*.
5. В текстовом окне Монитора должно появиться сообщение:

```
Welcome to SDT SIMULATOR. Simulating system
Demongame
```

При запуске исполняемой системы происходит создание статически заданных экземпляров процессов (в нашем случае `Main` и `Demon`), однако их стартовые переходы не выполняются.

Монитор готов к выполнению команд (при этом выдается приглашение `Command:` в текстовом окне).

10.3. Выполнение переходов

После выполнения упражнений данного раздела вы должны научиться следующему:

- вводить команды Монитора в текстовом виде;
- определять SDL-символ, который будет исполняться следующим;
- читать текстовые и графические трассы Монитора;
- выполнять очередной переход;
- пользоваться кнопками графического интерфейса Монитора;
- посылать сигналы из окружения системы.

10.3.1. Установка уровня детальности трассировки

В этом упражнении мы установим уровень детальности трассы, т.е. объем информации о ходе исполнения системы, который будет выдаваться Монитором. Для установки уровня детальности используется команда Монитора *Set-Trace*.

Будем вводить команды в текстовом виде в поле *Command*: (под текстовым окном сообщений Монитора).

1. Нажмите мышью в поле ввода команды. Введите команду *Set-Trace 6* и нажмите клавишу <Return>. Уровень детальности 6 означает полную информацию обо всех действиях, выполняемых на переходах. Введенная команда отображается в окне сообщений монитора. Монитор подтверждает установку нового уровня детальности трассы. Окно текстовых сообщений Монитора должно выглядеть следующим образом:

```
Welcome to SDT SIMULATOR. Simulating system
Demongame.
```

```
Command: set-trace 6
```

```
10.3.1.1.1 Default trace set to 6
```

```
Command:
```

2. Имеется возможность наблюдать выполнение системы графически, т.е. отображать исполняемые символы непосредственно на SDL диаграммах. По-умолчанию, графическая трассировка выключена. Для включения графической трассировки нужно выполнить команду *set-gr-trace 1*. Значение 1 указывает на то, что Монитор будет каждый раз показывать символ, который будет исполняться следующим.

10.3.2. Выполнение стартовых переходов системы

1. Введите команду *show-next-symbol*. Появится окно редактора SDL диаграмм, в котором будет показан стартовый символ процесса Main. В дальнейшем, это окно будет использоваться Монитором для показа графической трассы.
2. Выполните команду *next-transition* (введите *n-t* в поле ввода команды). Монитор отображает выполнение стартового перехода процесса Main двумя способами:
 - В окне сообщений Монитора текстовая трасса содержит следующую информацию о выполняемом переходе: идентификатор экземпляра процесса, выполняющего переход, имя исходного состояния, текущее значение модельного времени. Трасса перехода завершается строкой, содержащей имя следующего состояния.

```
*** TRANSITION START
*           Pid      :   Main:1
*           State    :   start_state
*           Now      :   0.0000
*** NEXTSTATE      Game_Off
```

- В окне редактора SDL диаграмм графическая трасса показывает символ, который будет исполняться на следующем переходе. Следующий символ – это стартовый символ процесса Demon .

3. Для выполнения стартового перехода процесса Demon воспользуемся возможностью интерфейса Монитора повторять уже введенные команды: нажмите мышью на поле ввода команд и нажмите клавишу <Up>. В поле ввода команды появится последняя введенная команда (*n-t*). Выполните ее, нажав клавишу <Enter>.

Выполняется стартовый переход процесса Demon. Обратите внимание на то, что редактор SDL диаграмм выделяет текстовый символ, содержащий определение таймера T. В системе SDT принято соглашение, что при отсутствии сигналов из окружения системы, следующим событием является срабатывание таймера.

Обратите внимание, что в текстовой трассе содержится сообщение об установке таймера T.

10.3.3. Посылка сигналов из окружения системы

Для взаимодействия с системой нужно уметь посылать сигналы из окружения системы (например, сигнал *Newgame*, предназначенный процессу *Main*.) Для посылки сигнала системе воспользуемся командой *Output-Via*, аргументами которой являются имя сигнала, параметры сигнала (отсутствуют в случае сигнала *Newgame*) и имя канала, по которому осуществляется посылка.

Для ввода команды воспользуемся возможностями графического интерфейса. Кнопки команд расположены на левой панели. Кнопки сгруппированы в соответствии с их назначением.

1. Выберите группу команд *Send Signal* и нажмите на кнопку *Send Via* (для нахождения этой кнопки может потребоваться прокрутить окно с кнопками). Данная кнопка выполняет команду *Output-Via* (выполняемая команда отображается в окне сообщений Монитора). Выполнение команды приводит к появлению нескольких дополнительных диалогов для ввода параметров команды. Первый диалог предназначен для ввода имени сигнала. Диалог содержит список всех сигналов, которые могут быть посланы системе из окружения.

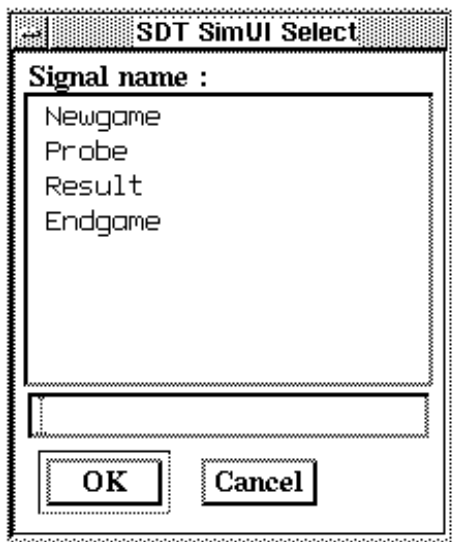


Рис. 54. Посылка сигнала *Newgame*

2. Выберите сигнал *Newgame* и нажмите кнопку *OK*.
3. Следующий диалог предназначен для ввода имени канала:

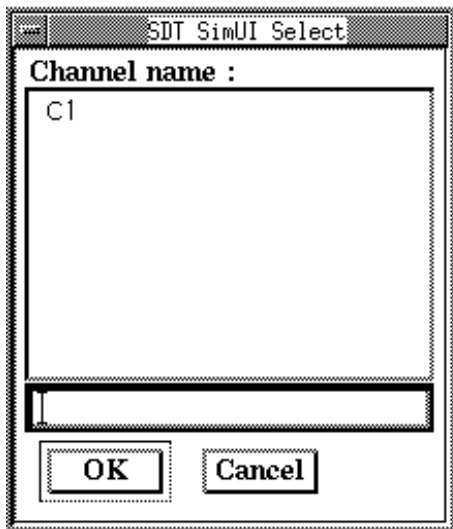


Рис. 55. Выбор канала для посылки сигнала

4. В списке имеется единственный элемент (канал C1). Выберите его, нажав на кнопку *OK*.

В окне сообщений подтверждается посылка сигнала. Графическая трасса показывает, что следующим символом является символ ввода сигнала Newgame.

10.3.4. Выполнение переходов

1. Выполните следующий переход, используя кнопку *Transition* в группе *Execute*. Текстовая трасса показывает действия, выполняемые на переходе в состояние Game_On. Заметим, что, в отличие от стартового перехода, начальное состояние перехода обозначается комбинацией имени состояния (Game_Off) и имени принимаемого сигнала (Newgame):

```
*** TRANSITION START
*           Pid       : Main:1
*           State     : Game_Off
*           Input     : Newgame
*           Sender    : env:1
*           Now       : 0.0000
*           CREATE    Game:1
*           ASSIGN    GameP := Game:1
*** NEXTSTATE   Game_On
```


На данном переходе создается экземпляр процесса Game, поэтому графическая трасса показывает, что следующим символом является стартовый символ процесса Game.

Таким образом, текстовая трасса показывает события, произошедшие на завершившемся переходе, включая исходное и заключительное состояния данного перехода. Графическая трасса показывает некоторое будущее событие, которое произойдет при выполнении следующего перехода (причем первый символ нового перехода может оказаться на диаграмме другого процесса).

2. Выполните очередной переход процесса Game, нажав на кнопку *Transition*. Процесс Game переходит в состояние *Losing* и графическая трасса возвращается в процесс *Demon*.

3. Выполните очередной переход (на котором должно произойти срабатывание таймера). На данном переходе таймер посылает сигнал процессу, который произвел установку данного таймера. Срабатывание таймера считается отдельным переходом. Обратите внимание на то, что на данном переходе произошло увеличение модельного времени.

```
*** TIMER signal was sent
*   Timer      : T
*   Receiver   : Demon:1
*** NOW       : 1.0000
```

4. Выполните очередной переход. Исходное состояние данного перехода характеризуется состоянием *Generate* и получением сигнала таймера *T*. На данном переходе происходит посылка сигнала *Bump* процессу *Game*:

```
*** TRANSITION START
*           Pid      : Demon:1
*           State    : Generate
*           Input    : T
*           Sender   : Demon:1
*           Now      : 1.0000
*   OUTPUT of Bump to Game:1
*   SET on timer T at 2.0000
*** NEXTSTATE  Generate
```

Графическая трасса показывает, что следующим символом будет прием сигнала *Bump* в процессе *Game*. Как видно на диаграмме, после получения

сигнала Bump процесс Game попадет в состоянии Winning, в котором будет ожидать приема сигнала Probe (либо нового сигнала Bump).

5. Выполните очередной переход для перевода процесса Game в состояние Winning. Графическая трасса возвращается в процесс Demon и показывает, что при отсутствии внешних сигналов следующим событием будет срабатывание таймера T. Вместо этого, пошлем сигнал Probe из окружения системы:
6. Пошлите сигнал Probe, используя кнопку *Send Via*. Графическая трасса возвращается в процесс Game.
7. Выполните следующий переход. На данном переходе происходит прием сигнала Probe и посылка сигнала Win в окружение системы. Процесс Game возвращается в состояние Winning и ждет сигнала Probe или сигнала Bump.

```
***  TRANSITION START
*          Pid      : Game:1
*          State    :  Winning
*          Input    :  Probe
*          Sender   :  env:1
*          Now      :   1.0000
*          OUTPUT  of Win to env:1
*          ASSIGN  Count := 1
***  NEXTSTATE Winning
```

10.4. Исследование внешнего поведения системы

После выполнения упражнений данного раздела вы должны научиться:

- перезапускать исполняемую систему;
- использовать возможность протоколирования внешних сигналов;
- добавлять новые кнопки на панель управления Монитором;
- выполнять все переходы в течение заданного интервала времени.

10.4.1. Протокол внешних сигналов

1. Перезапустите исполняемую систему: выберите команду *Restart* из меню *File* окна Монитора. Дополнительный диалог запросит подтверждения завершения текущей трассы. Нажмите кнопку *OK*. Окно сообщений

Монитора очищается и исполняемая система приводится в начальное состояние.

2. Установите уровень детальности трассы равный 1. Это означает, что трасса будет показывать только взаимодействие системы с окружением, т.е. сигналы из окружения и в окружение.
3. Для включения протокола внешних сигналов введите команду *signal-log* в поле ввода команды и нажмите клавишу <Return>. Для данной команды не существует стандартной кнопки на панели управления Монитором. Команда имеет два параметра, значения которых запрашиваются в дополнительных диалогах. Первый параметр – это имя SDL объекта, для которого нужен протокол сигналов. В окне представлен список всех объектов в системе.
4. Введите название объекта *env* в текстовое поле ввода в нижней части окна. Этот объект означает окружение системы.

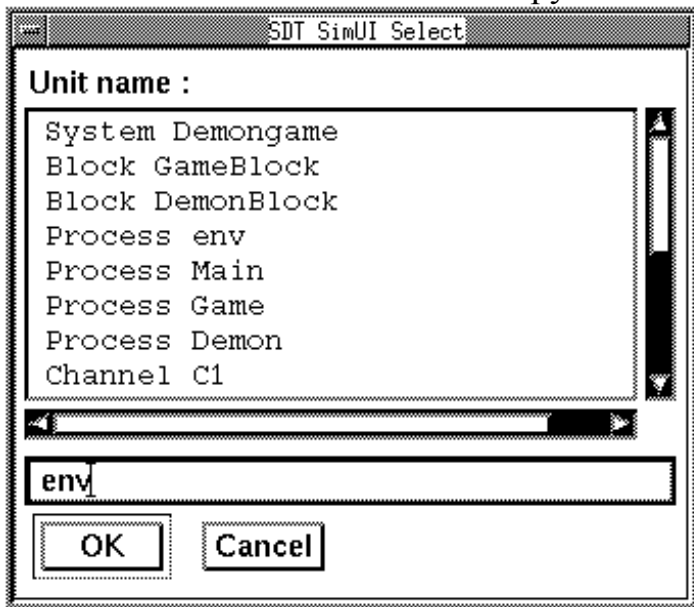


Рис. 56. Включение протокола внешних сигналов

5. Вторым параметром команды *signal-log* является имя файла, в который направляется протокол внешних сигналов. Появляется стандартный диалог выбора файла. Введите имя файла *signal.log* в поле *File*.

10.4.2. Добавление новых кнопок к интерфейсу Монитора

Графический интерфейс Монитора позволяет добавлять новые кнопки на панель управления Монитором и связывать с ними произвольные команды.

1. Выберите группу кнопок *Send Signal*. Выберите команду *Add* из меню *Group* для данной группы.

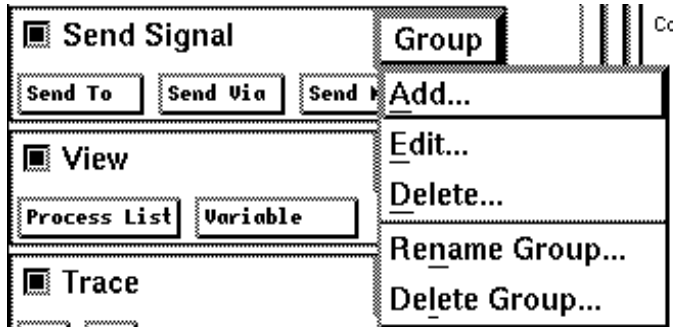


Рис. 57. Добавление новой кнопки к группе

2. В дополнительном диалоге введите *Newgame* как метку новой кнопки в поле *Label*. Не нажимайте клавишу *<Return>* !. Введите *output-via newgame -* в поле *Definition*. Последний параметр “-“ в команде *output-via* означает значение по-умолчанию (эквивалент нажатию на кнопку *OK*) в соответствующем диалоге.

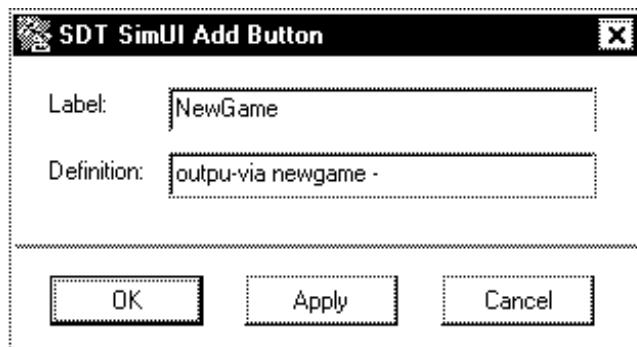


Рис. 58. Добавление кнопки

3. Завершите выполнение команды, нажав на кнопку *Apply*. Новая кнопка появляется в группе, диалог может быть использован для определения новой кнопки.
4. Добавьте кнопку для отправки сигналов *Probe*, *Result* и *Endgame*.
5. Завершите диалог, нажав кнопку *OK*.

10.4.3. Взаимодействие с системой

1. Пошлите сигнал *Newgame*, нажав на новую кнопку *Newgame*.

2. Выполните все переходы до достижения модельного времени равного 5.5. Для этого нажмите кнопку *Until Time* в группе *Execute*. Введите значение 5.5.
3. Пошлите сигнал *Probe*.
4. Выполните все переходы до достижения модельного времени 10.3. Обратите внимание на то, какой сигнал был послан системой в окружение (*Win* или *Lose*?).
5. Пошлите два сигнала *Probe* подряд.
6. Пошлите сигнал *Result*.
7. Выполните все переходы до достижения модельного времени 13.5. Обратите внимание на то, какие сигналы были посланы системой в окружение.

10.4.4. Просмотр протокола внешних сигналов

1. Завершите выполнение исполняемой программы, нажав кнопку *Stop Sim* в группе *Execute*. По этой команде выключится протокол внешних событий. Окно Монитора при этом остается активным.
2. Файл `signal.log` должен содержать протокол внешних сигналов исполняемой системы:

```
Signal log for system Demongame with unit Process env
on file ...
0.0000 Newgame from env:1 to Main:1
5.5000  Probe from env:1 to Game:1
5.5000  Win from Game:1 to env:1
10.3000 Probe from env:1 to Game:1
10.3000 Probe from env:1 to Game:1
10.3000 Result from env:1 to Game:1
10:3000 Lose from Game:1 to env:1
10:3000 Lose from Game:1 to env:1
10:3000 Score from Game:1 to env:1
Parameter(s) : 1
```

10.5. Генерация диаграмм взаимодействия

В данном упражнении мы рассмотрим способы автоматической генерации диаграмм взаимодействия по исполняемой системе. Монитор SDL позволяет представлять основные события SDL системы в виде событий диаграмм

взаимодействия. Это позволяет визуализировать трассу SDL системы в виде диаграммы взаимодействия.

После выполнения упражнений данного урока вы должны научиться:

- включать и выключать генерацию диаграммы взаимодействия;
- переходить от символов диаграммы взаимодействия к символам SDL;
- выполнять отдельный SDL символ.

10.5.1. Инициализация диаграммы взаимодействия

1. Перезапустите исполняемую систему.
2. Отключите графическую трассу для того, чтобы избежать появления окна редактора SDL диаграмм. Для этого нажмите на кнопку *SDL Level : Show* в группе *Trace* и убедитесь, что уровень детальности графической трассы (*GR Trace*) установлен в 0.
3. Для включения генерации диаграмм взаимодействия нажмите кнопку *MSC Trace : Start* в группе *Trace*. Выполнение данной команды приводит к появлению дополнительного диалога для задания уровня детальности диаграммы взаимодействия. Выберите 1 для генерации диаграмм взаимодействия с символами состояний. Завершите выполнение команды, нажав кнопку *OK*.

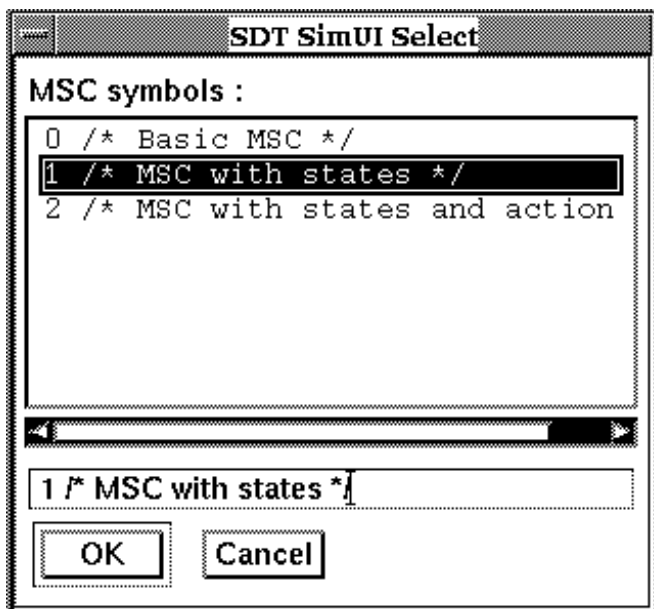


Рис. 59. Выбор уровня детальности диаграммы взаимодействия

4. Через несколько секунд появится редактор диаграмм взаимодействия, в котором открыта диаграмма с именем "SimulatorTrace". На диаграмме присутствуют три объекта: процессы Main_1_1, Demon_1_2 и процесс env_0 (представляющий собой окружение системы):

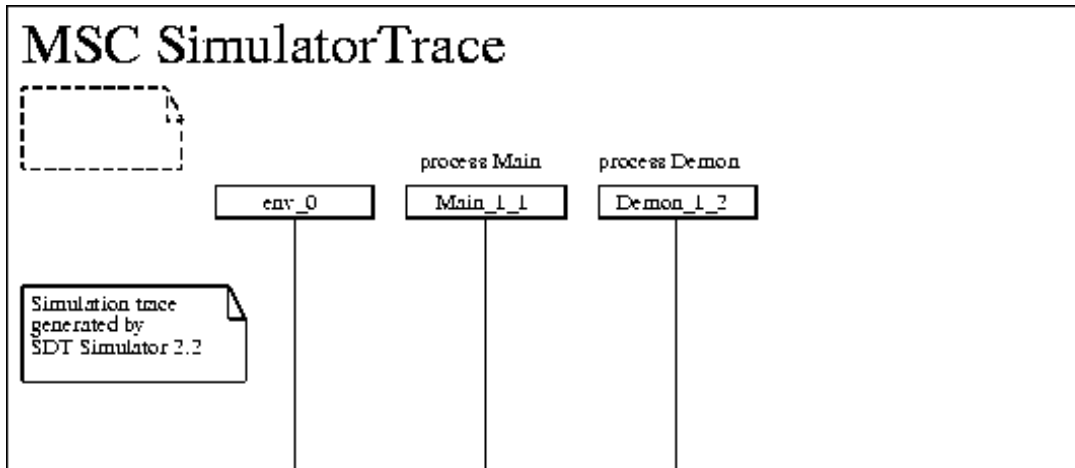


Рис. 60. Начальный вид диаграммы взаимодействия

10.5.2. Трассировка

1. Пошлите сигнал Newgame из окружения системы. Посылка сигнала представляется на диаграмме взаимодействия как сообщение от объекта env_0 к объекту Main_1_1. На данном этапе, сообщение помечено звездочкой на конце, означающей, что сообщение еще не принято процессом - получателем. (В ряде случаев появление звездочки может означать возникновение ошибки в спецификации, например, посылка сигнала несуществующему процессу). Заметим, что символ посылки сообщения, помеченный звездочкой, является расширением языка диаграмм взаимодействия, принятым в системе SDT.
2. Выполните очередной переход. На оси объекта процесса Main появляется символ состояния Game_Off. Данный символ означает, что процесс Main завершил переход и находится в соответствующем SDL состоянии.
3. Выполните очередной переход. На данном переходе процесс Demon устанавливает таймер T.
4. Выполните очередной SDL символ. Для этого нажмите на кнопку Symbol в группе Execute на панели управления монитора. Процесс Main принимает сигнал Newgame. На диаграмме взаимодействия это

представляется как снятие звездочки. Обратите внимание на то, что точки отправки и приема сообщения Newgame находятся на разных вертикальных уровнях, т.к. таймер T был установлен после того, как было послано сообщение.

5. Выполните следующий SDL символ (нажав на кнопку *Symbol* в группе *Execute*). Происходит создание экземпляра процесса Game. На диаграмме взаимодействия это представляется как появление нового объекта. Диаграмма взаимодействия на данный момент должна выглядеть следующим образом.

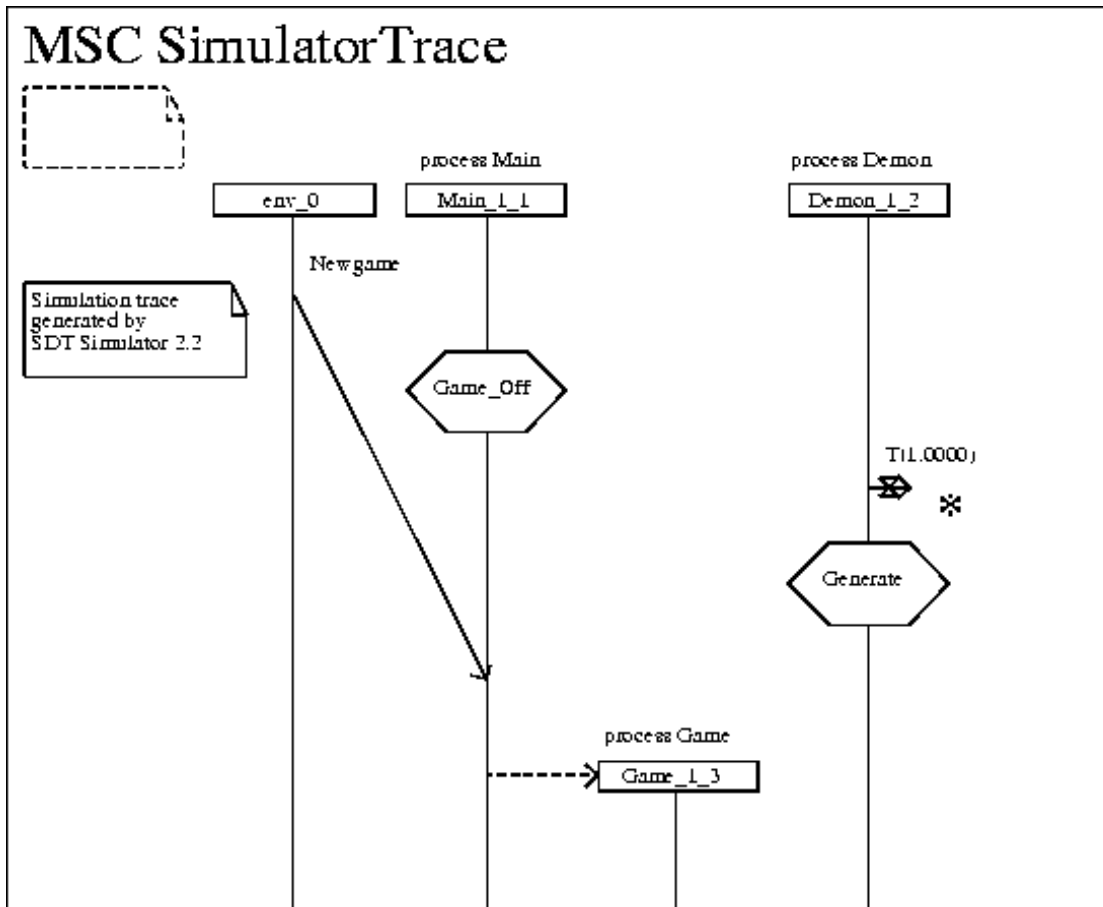


Рис. 61. Вид диаграммы взаимодействия после создания процесса Game.

6. Завершите текущий переход и выполните следующий переход, нажав два раза на кнопку *Transition*. Второй переход переводит процесс Game в состояние *Losing*.
7. Выполните следующие три перехода. Происходит прием сигнала таймера T, а также прием и посылка сигнала *Vump*. Процесс Game находится в

состоянии `Winning`. Обратите внимание на представление обмена сообщениями на диаграмме взаимодействия.

8. Покажем, как на диаграмме взаимодействия представляется немедленный прием сообщения. Пошлите сигнал `Probe` из окружения системы и выполните следующий SDL символ. Сначала появляется символ сообщения, помеченный звездочкой, а затем отметка снимается, но символ сохраняет горизонтальное положение.
9. Завершите текущий переход. Система посылает сигнал `Win` своему окружению.
10. Пошлите сигнал `Result` из окружения системы и выполните очередной переход. На диаграмме взаимодействия появится сообщение `Score` с параметром `1`.
11. Завершите игру, пошлав сигнал `Endgame` и выполнив два перехода. Происходит остановка процесса `Game`.

MSC Simulator Trace



Simulation trace generated by SDT Simulator 2.2

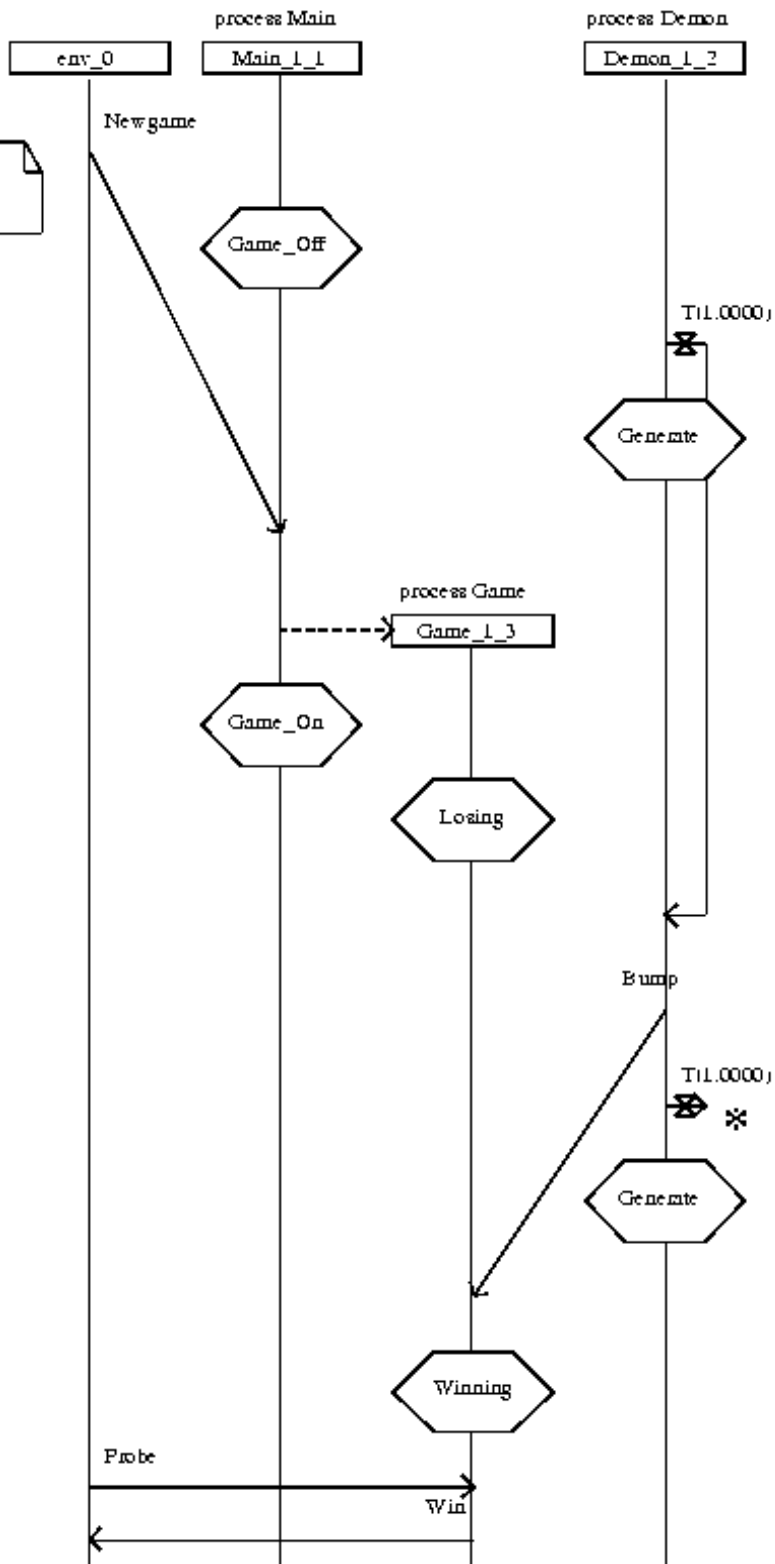


Рис. 62. Окончательный вид диаграммы взаимодействия (1)

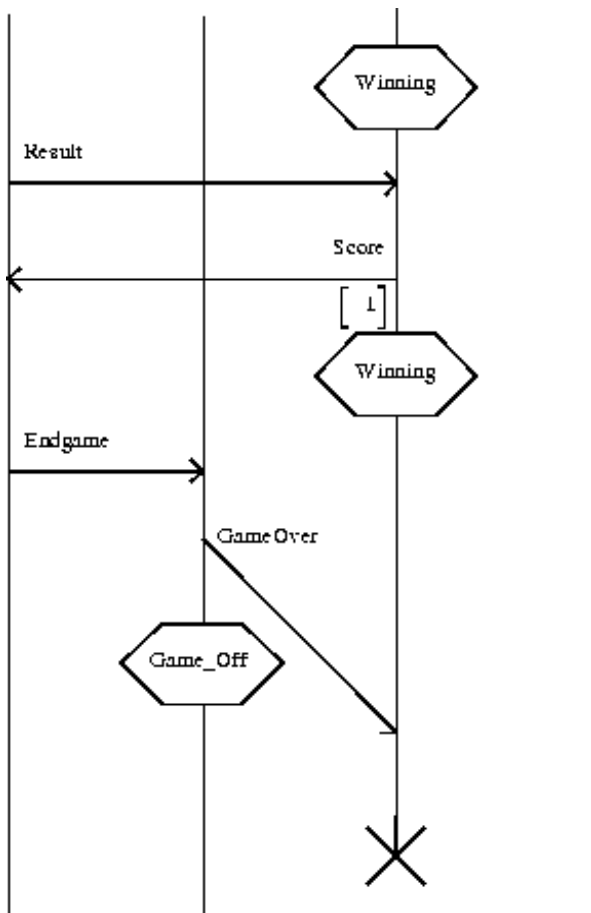


Рис. 63. Окончательный вид диаграммы взаимодействия (2)

10.5.3. Переход к SDL символам

1. На диаграмме взаимодействия щелкните мышью на символ, соответствующий сообщению `Winpr`.
2. Выберите команду *Info Window* в меню *Window* редактора диаграмм взаимодействия. Появляется окно с дополнительной информацией о выбранном объекте.

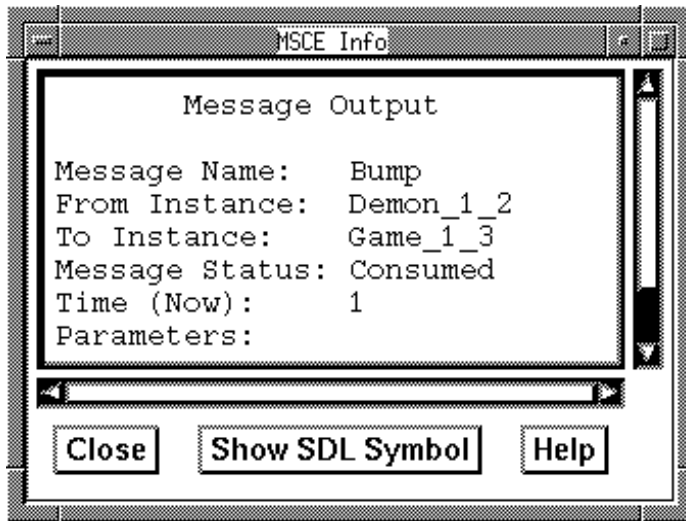


Рис. 64. Окно с дополнительной информацией о символе диаграммы взаимодействия

Нажмите на кнопку *Show SDL Symbol*. Появляется редактор SDL диаграмм с отмеченным символом послылки или приема сигнала *Bump*:

- Если щелчок мышью был произведен вблизи точки послылки сообщения, то будет показан соответствующий символ послылки сигнала;
- Если щелчок мышью был произведен вблизи точки приема сообщения, то будет показан символ приема сообщения.

10.5.4. Завершение трассировки

1. Завершите генерацию диаграммы взаимодействия выбрав команду *MSC Trace : Stop* в группе *Trace*.
2. Сохраните сгенерированную диаграмму взаимодействия в файле с именем *SimulatorTrace.msc*, выполнив команду *Save* из меню *File* редактора диаграмм взаимодействия.
3. Завершите работу с редактором диаграмм взаимодействия, выбрав команду *Exit* из меню *File*.

10.6. Заключение

В данной главе был описан процесс построения исполняемых систем по спецификациям и их выполнение под управлением Монитора. В результате выполнения упражнений данной главы вы должны были научиться:

- создавать исполняемую систему;
- вводить команды Монитора в текстовом виде;

- определять SDL-символ, который будет исполняться следующим;
- читать текстовые и графические трассы Монитора;
- выполнять очередной переход;
- пользоваться кнопками графического интерфейса Монитора;
- посылать сигналы из окружения системы.
- запускать Монитор исполняемой системы;
перезапускать исполняемую систему;
- использовать возможность протоколирования внешних сигналов;
- добавлять новые кнопки на панель управления Монитором;
- выполнять все переходы в течение заданного интервала времени.
- запускать систему из графического интерфейса Монитора;
- включать и выключать генерацию диаграммы взаимодействия;
- переходить от символов диаграммы взаимодействия к символам SDL;
- выполнять отдельный SDL символ.