Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Среда и язык реактивного программирования распределенных систем JARL

Магистерская диссертация
Соколова Николая Евгеньевича

Научный руководитель ….... Д.В. Луцив
ст. преподаватель / подпись /

Рецензент ….... В.В. Оносовский
с.н.с. лаб. информационных систем / подпись /

“Допустить к защите”
заведующий кафедрой, ….... А.Н. Терехов
д.ф.-м.н., профессор / подпись /

Санкт-Петербург
2012

Saint Petersburg State University
Mathematics & Mechanics Faculty

Software Engineering Chair

JARL Reactive Programming Language and Environment for Distributed Applications

by

Sokolov Nikolay Evgenievich
Master thesis

Supervisor
assistant professor
…..... D.V. Luciv
/ Signature /

Reviewer
senior research fellow, Information System lab.
…..... V.V. Onossovski
/ Signature /

“Admitted to proof”
head of Chair,
professor
…..... A.N. Terekhov
/ Signature /

Saint Petersburg
2012

Содержание

Введение....5
Постановка задачи....8
Глава 1. Обзор предметной области....9
Распределенные системы....9
CORBA....9
Enterprise JavaBeans....12
SOA....13
.NET Remoting....15
WCF....16
ZeroC ICE....16
XML-RPC....17
Erlang....17
Реактивное программирование....18
Yampa....19
FlapJax....22
Rx.NET....22
Графическое моделирование систем....23
SDL....23
Room....24
RTST....25
REAL....27
QReal....30
Глава 2. Jarl – Технология распределенного реактивного программирования.
....32
Подход к разработке....32
Терминология....34
Объекты конечной системы....35
Глава 3. Язык Jarl....36

Текстовая нотация....36
Графическая нотация....40
Глава 4. Реактивное программирование на языке Erlang....41
Поддержка состояний....41
Библиотека для событийно-ориентированного программирования....43
Библиотека для реактивного программирования....46
Маршрутизатор....48
Процесс работы с библиотекой....49
Глава 5. Генераторы кода....51
Генерация из текстовой нотации языка Jarl в Erlang....51
Генерация из текстовой нотации языка Jarl в Java....51
Генерация из модели диаграммы QReal в текстовую нотацию языка Jarl...53
Заключение....54
Возможности развития....54
Список литературы....56

Введение

Создание распределенных систем — специфичная область в сфере разработки программного обеспечения. Ко всем особенностям реализации приложения, расположенного на одной машине, добавляются новые: требуется учитывать возможную недоступность части машин, на которых исполняется приложение, большее время взаимодействия между компонентами системы. Поэтому для разработки распределенных систем уже долгое время существует ряд технологий и стандартов, упрощающих данный процесс. И они постоянно продолжают развиваться — не появился еще идеальный инструмент, который решил бы все проблемы. Примерами уже существующих инструментов и стандартов можно привести WCF (Windows Communication Foundation) [8], ZeroC ICE [9] и различные реализации CORBA [10] и XML-RPC[11].

При этом, представленные решения (за исключением, наверное, XML-RPC), стараясь достичь этого идеала, при ближайшем рассмотрении оказываются довольно громоздкими. Это приводит к увеличению ошибок как в области логики приложения, так и в организации коммуникаций между составными частями системы. Зачастую за большим объемом функциональности, предоставляемой в рамках данной технологии или подхода, теряется главное — взаимодействие между компонентами. Множественные уровни абстракции привносят в первую очередь дополнительные требования к конфигурации системы. Да, системы становятся намного более гибкими, но при этом для того, чтобы система функционировала положенным ей образом, приходится потратить немало усилий на ее первоначальную конфигурацию. Но и это не гарантирует то, что что она будет работать корректно — забытый параметр конфигурации может повлиять на работу системы самым неожиданным образом. Как следствие, вокруг таких технологий вырастает набор узких специалистов, которые с относительной легкостью могут настроить любое приложение, а у нового разработчика это зачастую вызывает затруднения.

Надо сказать, что данная ситуация в области разработки программного обеспечения характерна не только для распределенных систем. Различные технологии, библиотеки — они в первую очередь предназначены для облегчения работы программиста, но зачастую они делают обратное — под маской помощника скрывается монстр, разобраться с которым будет совсем непросто.

Отсюда напрашивается идея приближения программ, к таковым, написанным в стиле декларативного программирования. Существуют разнообразные методики достижения этой цели. К ним, например, относятся АОП (Аспектно-ориентированное программирование) [12], событийно-ориентированный подход к программированию или его потомок — реактивное программирование [14], [15],[16],[17],[18],[30], о котором и пойдет речь.

Основная идея реактивного программирования — организация направленных связей между компонентами приложения, причем таких, что при изменении значения компоненты, указанной в начале связи, в соответствии с некоторыми заданными правилами автоматически изменяется значение компоненты в конце связи.

Хороший пример применения реактивного программирования можно привести в области разработки пользовательских интерфейсов. Если реактивно привязать размеры и положение всех компонент интерфейса к размеру окна, мы при помощи совсем небольших усилий получим приложение с правильно масштабируемым графическим интерфейсом.

Идея данной работы — перенос положительного опыта применения реактивных систем на область разработки распределенных приложений.

И, действительно, в большинстве случаев организация частей приложений как сервисов, удаленный вызов процедур другой компоненты, просто не требуется, они служат только для одного — синхронизации данных между распределенными частями системы. Тогда почему же в данном случае не

применить так подходящее для этого реактивное программирование? Оно предоставляет возможность обеспечить согласованность соответствующих элементов удаленных компонент и позволяет при разработке отдельной компоненты не задумываться о наличием других компонент — все элементы являются локальными, а их автоматическое изменение переложено на среду Jarl.

Итак, целью данной работы является разработка языка Jarl и набора инструментов для определения реактивного взаимодействия компонент распределенного приложения.

Постановка задачи

- Изучение существующих технологий разработки распределенных и реактивных систем, а также средств их визуального моделирования;
- разработка нотаций языка Jarl:
  - текстовой,
  - графической;
- разработка библиотеки, обеспечивающей коммуникацию между компонентами распределенного приложения;
- разработка библиотеки, обеспечивающей реактивное связывание элементов компонент;
- разработка генераторов кода целевых языков;
- разработка графических редакторов языка Jarl;
- разработка генератора текстовой версии языка из графической.

Глава 1. Обзор предметной области

Распределенные системы

Простейший способ организации взаимодействия распределенных компонент — использование напрямую протоколов и механизмов стека TCP/IP [19],[20]. Но это значит, что разработчику потребуется разрабатывать протокол, заниматься организацией взаимодействия компонент. Причем для многих разрабатываемых приложений они практически не будут отличаться. Поэтому при разработке распределенных систем почти всегда используются уже готовые решения с более высоким уровнем абстракции. Ниже будут рассмотрены некоторые из них.

CORBA

Говоря о распределенных системах, надо начать в первую очередь со стандарта CORBA [10], как с одного из старейших используемых на данный момент.

Стандарт CORBA (Common Object Request Broker Architecture) появился в 1991 году.

И в первую очередь надо заметить, что CORBA — это прежде всего стандарт. Реализации же этого стандарта могут сильно различаться в зависимости от технологий, для которых они предназначены. Например, Inprise/Corel VisiBroker, Iona Orbix, Oracle Application Server реализовывают только часть функциональности, описанной в спецификации CORBA.

Программный код, согласно спецификации CORBA, объединяется в объект, содержащий информацию о самом коде, его функциональности и интерфейсах доступа к нему. Такие объекты могут вызываться из других программ или объектов CORBA, расположенных в сети.

Спецификация CORBA использует язык IDL [21], синтаксически похожий на язык описания классов в C++, для определения интерфейсов взаимодействия объектов с внешним миром.

На данный момент стандартизированы отображения из IDL в следующие языки: Ada, C, C++, Lisp, Smalltalk, Java, COBOL, Object Pascal, PL/1 и Python. Существуют также нестандартизированные отображения в Perl, Visual Basic, Erlang и TCL, реализованные для конкретных языков в виде брокеров (Object Request Broker, ORB) – программ-посредников, обеспечивающих удаленное взаимодействие распределенных компонент системы. Примерами таких брокеров являются:

- Borland Enterprise Server, VisiBroker Ed для Java и C++
- MICO для C++

В общем виде взаимодействие компонент, сгенерированного из них кода и брокеров показано на следующей диаграмме.

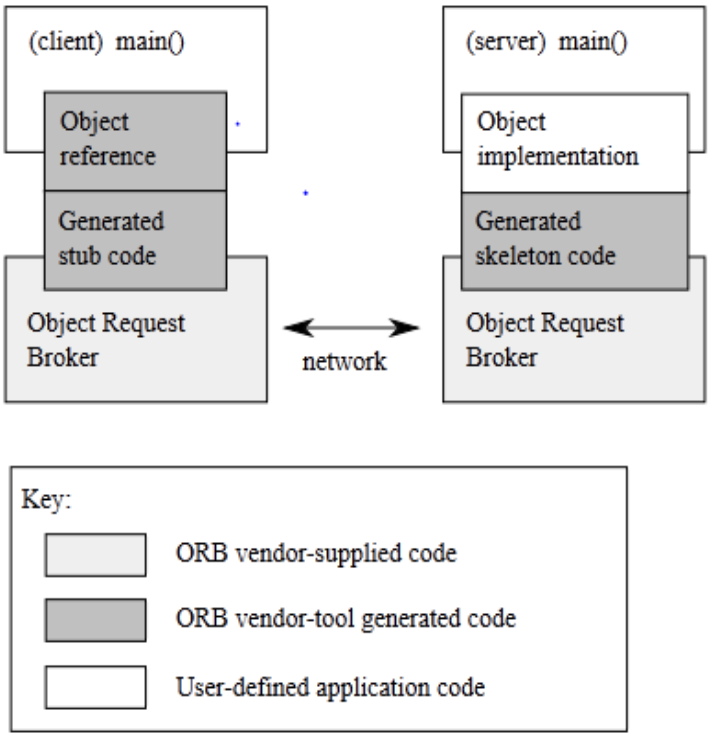


Рисунок 1: Взаимодействие объектов CORBA

В состав CORBA входит протокол GIOP (General Inter-ORB Protocol) – абстрактный протокол, обеспечивающий взаимодействие брокеров поверх различных протоколов. Его реализациями являются следующие протоколы.

- Internet InterORB Protocol (IIOP) используется для взаимодействия поверх протокола TCP/IP
- SSL InterORB Protocol (SSLIOP) — IIOP поверх протокола SSL. Поддерживает шифрование и аутентификацию.
- HyperText InterORB Protocol (HTIOP) — IIOP поверх протокола HHTP.

Протокол разрабатывается и поддерживается организацией OMG (Object Management Group).

Начиная с третьей версии в CORBA появилась возможность передавать объекты по значению. Это значит, что на противоположную сторону соединения передается объект целиком, со всеми его полями и методами, которые теперь будут исполняться локально.

Основная критика CORBA сводится к нескольким факторам.

- Большие временные промежутки между выходами новых версий.

  Это связано прежде всего с тем, что с тем, что новые стандарты CORBA принимаются организацией, состоящей из группы крупных IT-компаний. Каждая из них отчасти преследует свои интересы и продвигает свои технологии, что и замедляет принятие решений.

- Нестандартизованные отображения для большого количества языков и различные по поведению реализации брокеров.
- Сложность в освоении.

И именно последний недостаток сыграл решающую роль в судьбе CORBA. С появлением Java Enterprise Edition и Enterprise JavaBeans [22] технология CORBA отошла на второй план.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) [22] – технология для модульного проектирования и разработки информационных систем.

Была разработана в 1997 году в компании «Sun Microsystems».

Технология Enterprise JavaBeans в первую очередь предназначена для решения следующих задач:

- реализация транзакций, включающих взаимодействие одной или нескольких компонент,
- интеграция с различными сервисами, хранящими состояния компонент при помощи Java Persistence API,
- организация параллельного исполнение различных компонент, их взаимодействия и доступа к общим ресурсам,
- реализация событийно-ориентированного взаимодействия между компонентами,
- удаленный вызов методов (RPC),
- организация управления задачами,
- взаимодействие с удаленными компонентами (в том числе и с компонентами, реализованными в виде CORBA-объектов и веб-сервисов),
- обеспечение необходимого уровня безопасности системы,
- развертывание системы.

Первые версии EJB предоставляли только удаленный вызов методов при помощи CORBA, чем и навлекли себя довольно много критики. И уже с последующих версий функциональность стала расширяться не только в сторону обеспечения взаимодействия между компонентами, но и в сторону создания специальных механизмов для решения смежных задач. Это, скорее всего, и привело к большей популярности EJB по-сравнению с CORBA и некоторыми другими конкурентами.

SOA

Service-Oriented Achitecture (SOA) [25] – набор принципов и методологий организации приложения, как набора взаимодействующих сервисов.

Система при таком подходе разрабатывается как набор распределенных слабо связанных заменяемых компонентов с набором стандартизованных интерфейсов для взаимодействия при помощи стандартизованных протоколов.

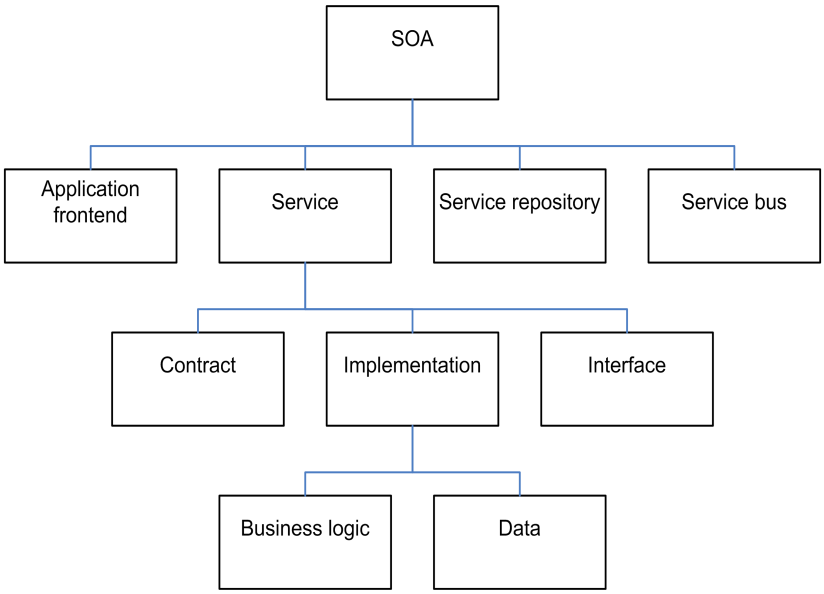


Рисунок 2: Элементы SOA

Интерфейсы компонентов в сервис-ориентированной архитектуре инкапсулируют детали реализации (операционную систему, платформу, язык программирования) от остальных компонентов, таким образом обеспечивая комбинирование и многократное использование компонентов для построения сложных распределённых программных комплексов, обеспечивая

независимость от используемых платформ и инструментов разработки, способствуя масштабируемости и управляемости создаваемых систем.

Хотя SOA не накладывает никаких ограничений на используемые языки и протоколы, для описания сервисов часто применяется язык WSDL (Web Services Definition Language) [26], а протокол SOAP (Simple Object Access Protocol) [27] используется как протокол взаимодействия.

Сервис в технологии SOA может выполнять две роли. Первая — зарегистрировать себя в реестре сервисов и опубликовать интерфейс доступа к себе. Вторая — выступить в роли клиентского сервиса. Клиентский сервис ищет необходимый сервис в реестре сервисов и при помощи заранее определенных механизмов получает доступ к найденному сервису через опубликованный им интерфейс.

За время существования SOA к нему сложилось неоднозначное отношение.

С одной стороны, SOA, предлагает некоторую альтернативу переиспользованию библиотек приложений — переиспользование сервисов. Причем это может быть не только повторное применение набора функций или методов, реализованных ранее, но и переиспользование логики приложений — ведь, как уже было сказано выше, одним из принципов SOA является выделение в сервисы слабо-связанных компонент, а они чаще всего и соответствуют некоторым объектам бизнес-логики. Таким образом, время и сложность разработки одного приложения может немного увеличиться, но при этом для каждого последующего схожего приложения они будут увеличиваться.

Также к плюсам SOA относят его релевантность бизнес-задачам, которые требуется решить. Так, например, для учета будет создан один сервис (или группа сервисов), для разработки документации — другой, и они внутри информационной системы будут общаться между собой через интерфейсы, схожие с бизнес-интерфейсами.

К недостаткам же SOA относят в основном необходимость следить за слабой связанностью модулей в процессе увеличения функциональности системы, сложность в тестировании и возможные проблемы с безопасностью у сервисов.

.NET Remoting

.NET Remoting [24] вместе с платформой .NET пришел на смену основанному на технологии COM DCOM. Он был выпущен в 2002 году компанией Microsoft и являлся реализацией концепции SOA для платформы .NET.

.NET Remoting позволяет создавать объекты специального типа (remotable object), доступ к которым есть у других процессов локальной или удаленной машины. При этом запросы к удаленным объектам для клиентского кода ничем не отличаются от локальных вызовов.

Взаимодействие с другими объектами производится по принципу запрос-ответ, и все запрошенные функции выполняются исключительно на серверной стороне.

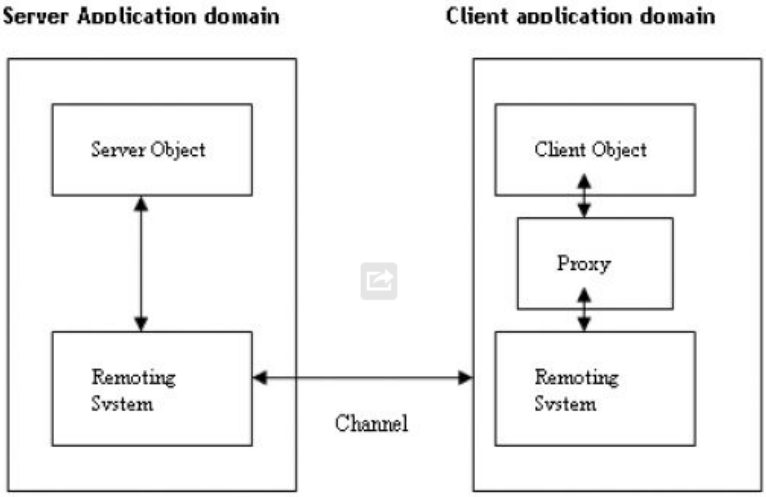


Рисунок 3: .NET Remoting

На рисунке 3 показано взаимодействие объекта-клиента и объекта-сервера. В момент обращения на клиенте создается класс-заместитель (proxy), дублирующий публичный интерфейс серверного объекта, с которым и производит общение объект-клиент и который передает все запросы объекту серверу, а также перенаправляет ответы.

WCF

Технология WCF [8], пришедшая вместе с .NET Framework версии 3.0 на смену .NET Remoting, является ещё одной реализацией концепции SOA от компании Microsoft.

Каждый сервис технологии WCF открывает «наружу» одну или более точку доступа к себе, каждая из которых содержит контракт, по которому следует обращаться к данному сервису.

Клиент, содержащий точку доступа, соответствующую точке доступа сервера (говорят, что точки доступа «связаны»), обращаясь по некоторому заранее заданному протоколу к точке доступа сервера, может, зная контракт, соответствующий данной точке доступа, получить необходимые ему данные.

Также для сервисов технологии WCF можно определить «поведения» — некоторый аналог аспектов [12] для сервисов. Они позволяют легко реализовать такие операции, как аутентификация пользователей, логгирование и др.

ZeroC ICE

ZeroC ICE [9] является ещё одной альтернативой таким крупным технологиям, как CORBA, EJB и SOA.

Все, что она предоставляет — удаленный вызов процедур (RPC).

Как и в CORBA, разработчик определяет набор методов, которые будет возможно вызывать удаленно из одного приложения в другое, и из этих методов генерируется интерфейсы взаимодействия на сервере и клиентах уже на целевых языках.

Одним из главных преимуществ ZeroC ICE является простота его использования о сравнению с другими известными технологиями.

XML-RPC

XML-RPC [11] — ещё одна спецификация, описывающая формат взаимодействия удаленных компонент системы при помощи вызова удаленных методов.

Спецификация данной технологии описывает исключительно протокол передачи данных.

Как и в случае ZeroC ICE, основным плюсом является простота, и именно из-за этого для большинства современных языков/платформ существует реализация XML-RPC.

Erlang

Говоря про организацию взаимодействия распределенных компонент приложения, нельзя не упомянуть язык Erlang [13]. Первая версия этого языка появилась в 1986 году в компании Ericsson, известной своими разработками в области телекоммуникаций.

Erlang – функциональный язык общего назначения , ориентированный в первую очередь на разработку параллельно-выполняющихся и распределенных систем.

Программа на языке Erlang – набор независимых потоков (процессов), способных асинхронно обмениваться сообщениями. Причем во время исполнения программы не делается никакого различия между потоками, выполняющимися на одной машине и на разных. Каждый процесс программы содержит «*mailbox*» — очередь сообщений, полученных процессом, но еще не обработанных. И эта очередь не является блокирующей, т.е. для обработки сообщения не требуется обработка всех предыдущих полученных сообщений — на следующем шаге обработки из очереди просто берется первое сообщение с существующим обработчиком, который и начинает исполняться. Такой механизм позволяет писать приложения с высокой скоростью обработки сообщений, что во многом и определило основную сферу применения языка

Erlang в области разработки программного обеспечения — высоконагруженные системы и системы, приближенные к системам реального времени.

В настоящий момент Erlang имеет внешние интерфейсы ко многим другим языкам: Java, C, C# (.NET), Python и др., что позволяет использовать его для организации сложных коммуникаций между распределенными компонентами, написанными на одних и тех же или разных языках программирования.

Реактивное программирование

Реактивное программирование можно определить как программирование в терминах потоков данных.

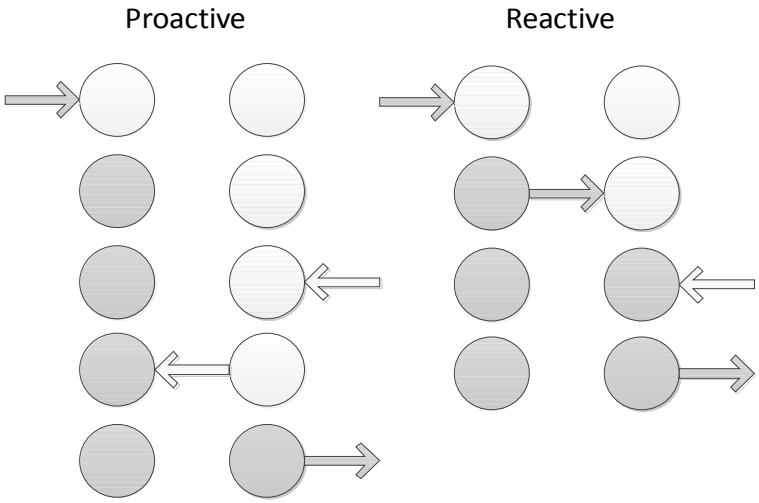


Рисунок 4: Сравнение проактивного и реактивного подходов

Приведем пример, показывающий различия классического (проактивного) подхода и реактивного.

Пусть у нас есть два поля: левое и правое, причем значение правого поля всегда есть значение некоторой функции, примененной к левому полю.

Пусть в какой-то момент времени значение левого поля изменилось. Тогда в случае проактивного подхода, когда нам понадобится вычислить значение правого поля, придется пересчитать его значение, а затем уже вернуть.

При реактивном же подходе значение правого поля изменится автоматически при изменении левого, и все, что нам останется — только считать его значение.

В большинстве существующих реализаций реактивный подход реализуется на базе событийно-ориентированного.

Yampa

Yampa [28] — функциональный реактивный язык, разработанный в Йельском университете.

В основе языка Yampa лежат два типа объектов.

- Сигнал — функция преобразующая точку во временной оси в определенный объект.
  `Signal 'a : Time → a`
  `a` – тип рассматриваемого объекта,

  `Time` – тип, соответствующий точке на оси времени. Время непрерывно, и представляется положительным числом.

  Например, если рассматриваемый объект — положение указателя мыши на экране, то реактивно зависимое от времени положение указателя мыши будет задаваться типом Signal Point.

- Сигнальная функция — функция для преобразования сигналов.
  `SF 'a 'b : Signal 'a → Signal 'b`

Сигналы и сигнальные функции можно наглядно представить в виде потоковой диаграммы. На ней сигналам буду соответствовать связи между объектами, а сигнальным функциям — сами объекты.

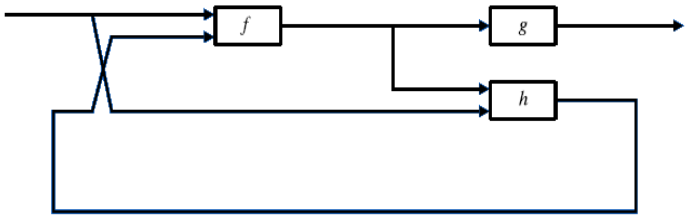


Рисунок 5: Пример диаграммы Yampa

Для работы с сигнальными функциями вводится несколько дополнительных функций.

- `arr :: ('a → 'b) → SF 'a 'b`

  Функция arr “поднимает” обычную функцию до сигнальной.

  Приведем пример. Пусть имеется функция, считающая для точки координаты точки, отстоящей от нее на 100 пикселей по-горизонтали. Тогда “поднятая” функция будет автоматически преобразовывать полученный на вход поток состояний точек (сигнал) в новый сигнал — поток сдвинутых точек. Таким образом из “обычной” функции была получена реактивная (в Yampa – сигнальная).

- Функции композиции:

```
(<<<) :: SF 'b 'c → SF 'a 'b → SF 'a 'c
(>>>) :: SF 'a 'b → SF 'b 'c → SF 'a 'c
```

- И много других функций. Ниже представлены некоторые из них в графической нотации языка Yampa.

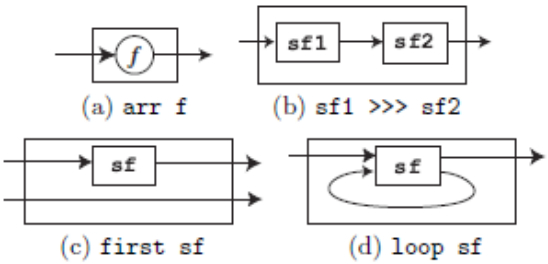


Рисунок 6: Некоторые функции языка Yampa

Текстовая нотация языка Yampa обладает синтаксисом, схожим с таковым в графической нотации. Здесь используется нотация Патерсона (Paterson's arrow notation) [36], транслирующаяся при помощи препроцессора в стандартный код языка Haskell [37].

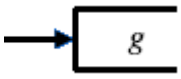
Приведем пример кода, написанного в данной нотации.

```
sf = proc (a,b) -> do
       c1 <- sf1 -< a
       c2 <- sf2 -< b
```

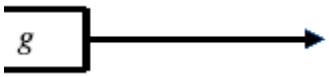
```
c <- sf3 -< (c1,c2)
rec
   d <- sf4 -< (b,c,d)
returnA -< (d,c)
```

В данном отрывке программы на языке Yampa описывается новая сигнальная функция, принимающая на вход пару сигналов, и возвращающая новую пару. Рассмотрим это определение подробнее.

- Конструкция “`proc .. -> do .. returnA`” просто задает анонимную сигнальную функцию.
- Оператор “`-<`” задает передачу сигнала на вход сигнальной функции. В графической нотации этому оператору соответствует следующий элемент:



- Оператор “`<-`” задает связь от выхода сигнальной функции к сигналу и в графической нотации изображается следующим образом:



Выше были приведены примеры работы с непрерывными потоками событий, но стоит также отметить, что Yampa поддерживает и дискретные события. Тип события определяется похоже на классический для функционального программирования тип Maybe.

```
Event 'a : NoEvent | Event 'a
```

Но их обработка производится не совсем классическим методом. Поскольку Yampa является функциональным языком, он не поддерживает состояния, соответственно обработчик не может изменить, как это происходит в большинстве событийно-ориентированных языках, состояние системы. Здесь можно задать непрерывную сигнальную функцию, которая по наступлении некоторых дискретных событий может менять свое поведение.

```
switch :: SF 'a ('b,Event 'c) → ('c->SF 'a 'b) → SF 'a 'b
```

Функция switch получает на вход два аргумента.

- Сигнальную функцию, дискретизирующую поток событий, т.е. преобразующую непрерывный поток значений в два новых потока: поток измененных значений и поток значений типа `Event 'c`
- Обработчик наступившего события, изменяющий выходную сигнальную функцию.

FlapJax

FlapJax [29] – ещё один функциональный реактивный язык, предназначенный прежде всего для разработки клиентской части веб-приложений.

FlapJax предоставляет возможности для

- событийно-ориентированного программирования
- реактивного программирования
- работы с потоками событий

FlapJax основан на языке JavaScript, что позволяет ему, во-первых, работать в большинстве современных браузеров и, во-вторых, взаимодействовать с кодом, написанным на языке Javascript.

В целом, идея, лежащая в основе FlapJax, похожа на рассмотренную ранее в Yampa. Здесь также есть непрерывные и дискретные потоки событий (правда, они не дифференцируются так сильно, как в Yampa). Такие потоки в языке FlapJax называются поведениями (behaviours). Здесь также можно “поднять” функцию для того, чтобы она получила возможность изменения потоков событий.

Rx.NET

Библиотека Rx [31],[32] для платформы .NET позволяет работать с наборами событий, как с потоками событий, одновременно предоставляя разработчику всю мощь LINQ.

Так потоки событий можно фильтровать, их можно сливать с один поток и делать практически все то, что можно делать со списками. И, при этом, на потоки событий можно подписываться как на простые события.

Являясь скорее расширением событийно-ориентированной парадигмы, библиотека Rx тем не менее подходит под описанное выше определение реактивного программирования. Изначальный поток, полученный из события, автоматически изменяется по приходу нового события, а вслед за ним изменяются и все потоки, полученные из него. Тем более, ранее говорилось, что почти всегда реализация реактивных компонент системы базируется на событийно-ориентированном подходе.

Графическое моделирование систем

SDL

SDL [33],[34] разрабатывался как язык для спецификации поведения реактивных и распределенных систем. Поддерживается организацией ITU-T в соответствии с рекомендациями Z.100 [33] и Z.106.

SDL является обобщением языка блок-схем с возможностью специфицировать структурную декомпозицию разрабатываемой системы, описывать параллельные процессы и их поведение.

Язык содержит две нотации.

- SDL/GR – графическая нотация.

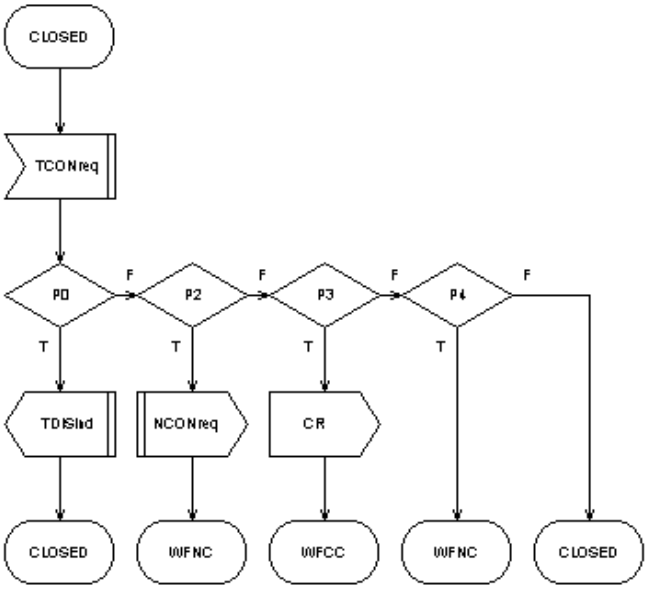


Рисунок 7: Пример SDL-диаграммы

- SDL/PR – текстовая нотация.

Но чаще всего употребляется графическая нотация, а текстовая используется в основном для переноса спецификаций из одного инструмента в другой. Ниже приведен пример диаграммы на языке SDL.

Room

Real-Time Object Oriented Modeling (ROOM) [35] – методология, предназначенная для разработки систем реального времени. Основным принципом методологии является использование одних и тех же диаграмм (моделей) на всех этапах разработки продукта.

Модели ROOM состоят из актеров, взаимодействующим друг с другом посредством пересылки сообщений по каналам, называемым «протоколами». Сами актеры тоже могут рассматриваться как подмодели и быть описаны таким же образом. Также поддерживается наследование актеров. Пример ROOM-модели представлен ниже.

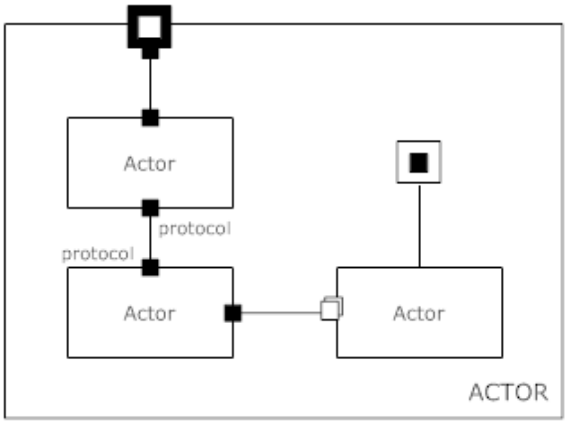


Рисунок 8: Пример ROOM-модели

RTST

RTST (Real-Time Software Technology) [2] — ещё одна технология, предназначенная для разработки систем реального времени. Была разработана в лаборатории системного программирования ВЦ ЛГУ (совр. СПбГУ) для разработки программного обеспечения телефонных станций телефонных станций.

В RTST были реализованы основные графические конструкции конечного автомата SDL, а для их “заполнения” использовался язык Алгол 68.

Разработка системы с помощью технологии RTST начинается с описания на специальном языке схемы объектов. Сначала объект описывается как черный ящик (точки его подключения к другим объектам, перечень входящих и исходящих сообщений и их параметры). Затем описываются внутренние атрибуты объекта.

С помощью транслятора схем такие спецификации объектов преобразуются во внутреннюю схему данных.

Поведение объекта специфицируется с помощью расширенного конечного автомата языка SDL/GR [33], позволяющего специфицировать структурную

декомпозицию разрабатываемой системы, описывать параллельные процессы и их поведение.

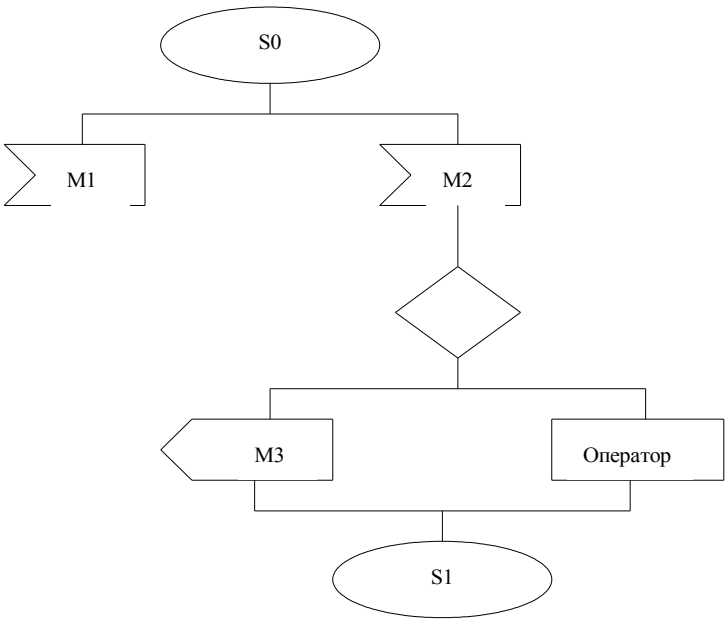


Рисунок 9: Пример диаграммы

С помощью конвертора SDL-диаграммы превращаются в текст на алгоритмическом языке (Алгол 68 и, с определенного момента, С), при этом происходят дополнительные проверки на соответствие со схемой объектов, по которой также происходит генерация служебных процедур (посылка сообщения, генерация экземпляра объекта в оперативной памяти и т.д.).

Тексты, полученные в результате конвертации, транслируются в коды целевой управляющей ЭВМ (или, в целях отладки, в коды инструментальной ЭВМ) и собираются вместе со средствами динамической поддержки в загрузочный модуль.

Настройка типовой программы на конкретное приложение осуществляется посредством формирования статической базы данных, содержащей описание конфигурации аппаратных средств встроенной системы и ее внешней среды.

Поддержка функционирования объектов осуществляется диспетчером, который производит последовательный запуск программ поведения готовых к исполнению объектов и управляет обменом сообщениями между связанными

объектами. По запросам от СУБД диспетчер осуществляет генерацию и уничтожение объектов, запуск и остановку их программ поведения, соединение и разъединение.

REAL

REAL — CASE-пакет языка визуального моделирования для проектирования компонентного ПО со сложной событийно-ориентированной логикой и возможностью автоматической генерации конечного кода. Как и RTST, он был разработан в лаборатории системного программирования НИИИТ СПбГУ совместно с кафедрой системного программирования математико-механического факультета СПбГУ и ЗАО “Ланит-Терком”.

Для описания объектов технологии REAL используется несколько расширенный набор уже известных языков описания диаграмм SDL/GR, Room, UML. Причем они описывают существующую систему и ее компоненты с разных сторон, что позволяет более полно графически описать ее поведение и структуру. Также, по сравнению с RTST, это предоставляет возможность описания более широкого, чем системы реального времени, класса систем.

Компонентная модель REAL состоит из трех типов объектов

- Компонента
- Интерфейс
- Порт

Компонента – это независимый элемент ПО, скрывающий свою реализацию и взаимодействующий с внешним миром через интерфейсы. Компонента может быть переиспользована в различных приложениях и развиваться независимо от использующего ее окружения. Кроме того, она может тиражироваться, переиспользоваться и заменяться другими компонентами, имеющими тот же внешний интерфейс.

Интерфейс – это описание правил взаимодействия между компонентами.

Интерфейс описывает правила взаимодействия двух объектов, которые его поддерживают. Он может содержать:

- набор сообщений с параметрами и пометкой направления;
- набор методов с параметрами и пометкой направления;
- набор атрибутов с пометкой направления и правами доступа (чтение, изменение);
- протокол, описанный при помощи диаграмм взаимодействий

Интерфейсы подключаются к компонентам через порт. В один порт может быть подключено много интерфейсов. Порт может моделировать множественное подключение. Один и тот же интерфейс может быть подключен к разным портам одной и той же или разных компонент и т. д.

Порт – это носитель конкретного соединения со стороны содержащей его компоненты. В простейшем случае порт может быть реализован как указатель на второго участника взаимодействия, в более сложном случае он может содержать специальные данные, методы и таймеры для контроля целостности соединения и обработки данных, поступающих по интерфейсу. По сути, эти элементы являются членами компоненты, которая содержит порт, но они выделяются в сам порт, если используются только для данного соединения.

Интерфейс описывает взаимодействие двух анонимных сторон. Для того, чтобы определить возможность взаимодействия между классами, интерфейс нужно связать с портами этих классов. Причем, с одним из портов интерфейс связывается как “прямой”, а с другим – как “обратный”.

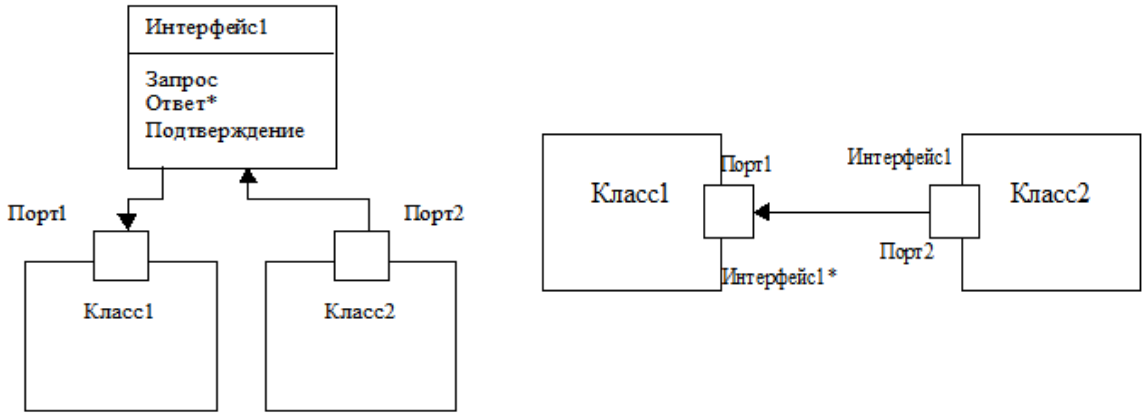


Рисунок 10: Компонентные диаграммы REAL

Поведенческая модель REAL – это специфический взгляд на поведение компоненты, основанный на понятиях *состояние*, *событие*, *переход*, *действие*. Данная поведенческая модель имеет две графических нотации: одну, основанную на STD, другую – на SDL.

Состояние – это период жизни компоненты, когда она готова принять запрос на взаимодействие от других компонент. Состояние может быть сложным, т.е. содержать другие состояния. С состоянием могут быть связаны действия компоненты. В этом случае состояние используется и как средство декомпозиции поведения компоненты.

Действие – это средство спецификации элемента поведения компоненты, вынесенного на уровень схемы. Действие может определяться либо средствами модели (посылка сообщения, установка таймера и т.д.), либо быть вставкой фрагмента на языке реализации, может быть связанным либо с состоянием, либо с переходом.

Переход – это линейная последовательность действий, осуществляемых компонентой вне состояния. В отличие от UML, переходы в REAL, как и в SDL, не мгновенны. Переход компоненты из состояния в состояние может включать в себя несколько переходов поведенческой модели REAL, связанных друг с другом через завершители различных типов – логическое ветвление, метку и т.д.

Событие – это единственный способ взаимодействия компонент REAL. События могут создаваться либо самими компонентами, либо системой поддержки поведенческой модели уровня реализации (таймерные события).

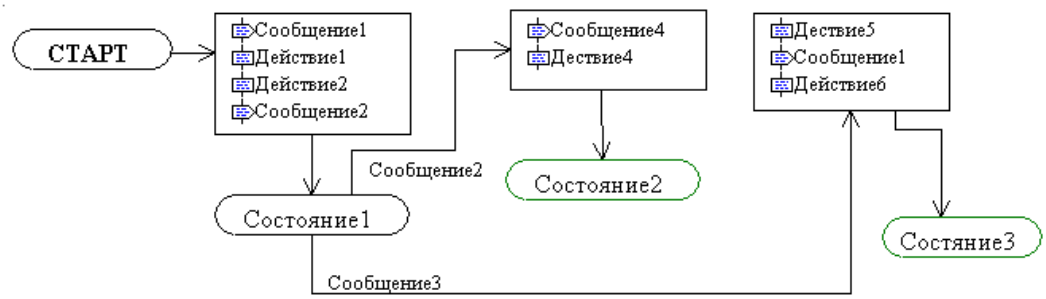


Рисунок 11: Пример STD-нотации REAL

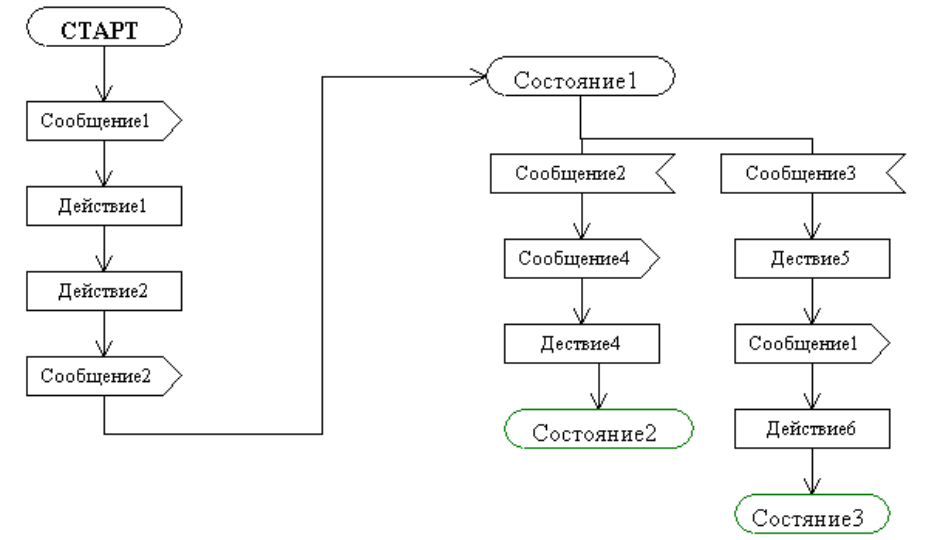


Рисунок 12: Пример SDL-нотации REAL

QReal

QReal [4] — MetaCASE-система для создания визуальных редакторов. Кроме метасредств, эта система включает в себя также реализацию редакторов для основных диаграмм UML 2.1 [38] и некоторых других видов диаграмм.

Разработкой технологии QReal занимается научно-исследовательская группа изучения технологий визуального моделирования кафедры системного

программирования Санкт-Петербургского Государственного Университета под руководством проф. А.Н. Терехова.

В QReal реализованы два альтернативных подхода, позволяющие пользователю системы создать новый редактор диаграмм, встраиваемый в QReal.

- Метамодель разрабатываемого языка описывается в виде XML-формата довольно простой структуры. Графические изображения элементов задаются с помощью текстового языка SDF, являющегося расширением языка описания векторной графики SVG 4.
- Метамодель языка задается графически в метаредакторе QReal посредством простого визуального языка, являющего аналогом MOF [39].

Для описания представлений элементов языка на диаграммах используется графический редактор форм, позволяющий создавать из набора примитивов векторные изображения или загружать уже готовые растровые.

Стоит отметить, что в QReal эти два подхода являются взаимозаменяемыми, поскольку XML-описание метамодели языка может быть как загружено в метаредактор, так и сгенерировано из него.

В дополнение к метаредактору для быстрого создания трансляторов визуальных диаграмм в исходный код на некотором текстовом языке в QReal используется специальный язык описания генераторов. Так, для каждого разрабатываемого визуального языка можно задать правила обхода созданных с его помощью диаграмм и генерации кода по ним. В дальнейшем эти правила интерпретируются для конкретных диаграмм, порождая соответствующий им код на выбранном целевом языке.

Глава 2. Jarl – Технология распределенного реактивного программирования.

Технология Jarl является продолжением реактивного подхода к программированию в области разработки распределенных систем.

Подход к разработке

Технология Jarl в первую очередь предназначена для создания скелета системы и определения реактивных связей между ее компонентами, не затрагивая их конкретные реализации. Это позволяет делать компоненты заменяемыми, облегчает их разработку и поддержку.

Разработку системы с использованием Jarl можно разделить на следующие стадии:

- разработка графической модели системы в графической нотации языка Jarl,
- генерация кода в текстовой нотации языка Jarl,
- генерация кода целевых языков и коммуникационных компонент,
- реализация конкретных компонент системы.

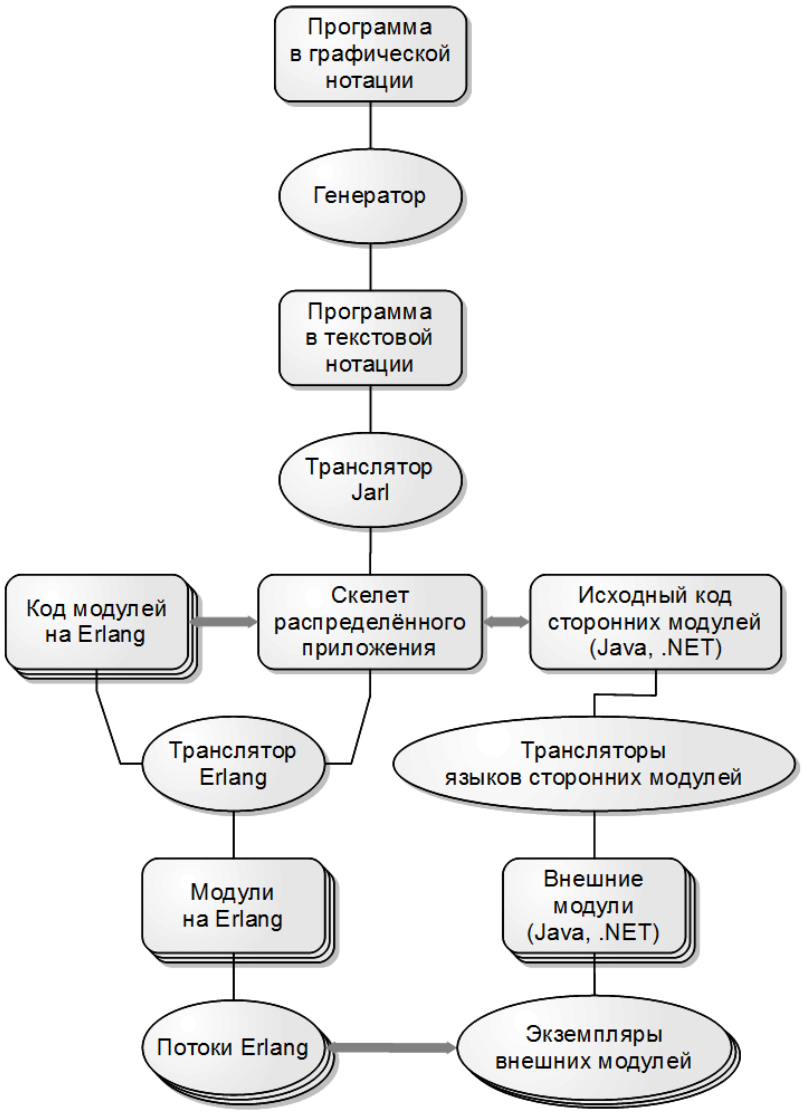


Рисунок 13: Этапы разработки

Стоит отметить, что первый шаг является необязательным, текстовая нотация языка Jarl несколько шире графической, правда она в то же время является менее наглядной.

Для дальнейшего описания технологии Jarl, потребуется ввести некоторые определения.

Терминология

В основе Jarl, как и многих других технологий для разработки распределенных систем, лежит понятие модуля. Его аналогами являются: объект в CORBA, bean в Enterprise Java Beans, сервис в SOA (WCF). *Модуль* — изолированная функциональность приложения, производящая общение с другими модулями строго через предназначенные для этого технологией Jarl механизмы. Модуль может быть описан, как на одной из нотаций языка Jarl, так и на любом другом языке, для которого реализован внешний интерфейс к Jarl.

Модули делятся на два типа: Jarl-модули и сторонние модули. *Сторонние модули* — абстрактные модули, для которых должен быть сгенерирован только интерфейс доступа. Они будут реализованы уже после генерации всех модулей в соответствии с требованиями к функциональности разрабатываемой системы. *Jarl-модули* — модули, как следует из названия, написанные на одной из нотаций языка Jarl и чаще всего предназначенные для организации коммуникации между другими модулями. Надо отметить, что, если требуется организация более сложного взаимодействия между модулями, возможно описание сторонних модулей на языке Jarl. Но тогда потребуется дополнительный этап генерации этого модуля уже после создания скелета основной системы. Поэтому удобнее описывать такие модуле на в виде сторонних модулей, а как Jarl-модули основной Jarl-программы.

Модули в программе на языке Jarl образуют иерархию, где на первом уровне находится Jarl-модуль. Модуль, включающий в себя другие модули, называется *родительским* по отношению к этим модулям.

Общение между модулями производится только через заранее определенные точки доступа к модулям — *порты.* Они являются аналогами портов в таких технологиях, как ROOM или REAL. Это именно те конструкции Jarl, которые можно реактивно связывать. Причем эта связь может быть, как между портами одного модуля, так и между портами модулей, находящихся внутри одного модуля, или портами модуля и его подмодулей.

Порты являются частными случаями *полей* модуля, типизированных объектов, хранящих некоторые значения. Порты — поля, открытые для доступа (чтения или записи) для родительского модуля.

Объекты конечной системы

После выполнения всех описанных выше действий на руках разработчика будут следующие объекты:

- модули на языке Erlang, соответствующие модулям языка Jarl,
- элементы программ на целевых языках сторонних модулей, предоставляющие интерфейс для взаимодействия с другими модулями,
- вспомогательные модули на языке Erlang для каждого из сторонних модулей.

О них подробнее будет рассказано в последующих главах.

Глава 3. Язык Jarl

Язык Jarl содержит две нотации: текстовую и графическую.

Текстовая нотация

Текстовая нотация языка Jarl поддерживает как возможность создания реактивных связей между полями модулей, так и возможность написания событийно-ориентированных программ.

Ниже приведен листинг программы, написанной в текстовой нотации языка Jarl и содержащий все конструкции, поддерживаемые текстовой нотацией языка Jarl.

```
s0 <- module()
      [params
         node:   "node0@JentalNote";
         cookie: "test";
      ]
           {
             a <- 0;
             b <- 0;
             c <- 0;

             sm1 <- module()
                    [params
                  node: "node1@clouded";
                  name: "javaModule0";
               ]
               [foreign
                  language: "Java";
                  in:       [{a, Integer}];
                  out:      [{c, Integer}];
               ]
               [java
                  package: "rl_test";
               ];

             f1 <- fun(x, y, z)
                   {
                     x + y + z;
                   };

             h1 <- handler
                   {
                     c <- new(a);
                   };

             e1 <- event
                   [on
                     condition : (a = 10) & (b < 5);
                   ],
                <@ h1,
                <@ handler
                   {
                     c <- 10;
                   };

             e2 <- !(d = 5);
             e3 <- event(d = 4);
             h2 <- #{d <- 0; a <- 8;};

             !(a = 5) <@ #{c <- 5;};

             sm1.a <~ a + 1;
             a <~ f(b, c, d);

             f2 <- fun(x, y)
                   {
                     {x, x + y};
                   };
             {a, b} <~ f2(a, c)
           };
```

Теперь рассмотрим по-очереди все конструкции языка.

Оператор “<-” — оператор присваивания. Так модуль задается присваиванием некоторому полю значения типа “модуль”, создаваемого при помощи ключевого слова “module”. Причем основной модуль программы тоже определяется через присваивание к некоторому полю, не принадлежащему никакому модулю.

При определении модуля можно указать некоторое количество атрибутов, уточняющих свойства или поведение модуля. Каждый атрибут должен быть заключен в квадратные скобки. Так для основного модуля требуется указать только название вычислительного узла Erlang, на которой он будет исполняться и cookie, используемый для построения сети Erlang. Для стороннего же модуля, который будет написан на Java, можно не указывать cookie в атрибуте params, зато требуется указать два дополнительных атрибута, один — обязательный для всех модулей, другой — только для модулей, которые будут реализованы на языке Java.

Тело модуля определяется внутри фигурных скобок и может содержать набор выражений. В число таких выражений входит и присваивание. Выражение представляет собой объект, к которому применяется некоторое действие, и список пар, разделенных запятой и состоящих из бинарного оператора и выражения, которое должно быть применено к объекту. Приведем пример.

```
e1 <- event
              [on
                condition : (a = 10) & (b < 5);
              ],
        <@ h1,
        <@ handler
               {
                 c <- 10;
               };
```

Здесь сначала полю e1 присваивается значение-событий, а затем к этому полю (не событию) привязываются два анонимных обработчика.

В примере выше также была показана возможность присваивания полям значений других типов. Полям также можно назначать значения типов “обработчик события”, “функция”.

На самом деле событие (как и обработчик) не обязательно должно быть присвоено полю — можно создать анонимное событие и привязать его к анонимному же обработчику.

Событие задается при помощи ключевого слова “`event`” и имеет обязательный атрибут “`on`”, в котором указывается логическая функция, которая инициирует событие. Для выражения, создающего событие, существует сокращение — ключевое слово “`event`” (или “`!`”), после которого в скобках указан параметр — логическая функция, которая инициирует событие.

Обработчику же событий, создающемуся при помощи ключевого слова “`handler`”, не требуется указывать никаких дополнительных атрибутов, но, в отличие от события, он имеет тело, в котором указывается, какие действия требуется совершить. Для обработчика тоже существует сокращенный способ записи — ключевое слово “`handler`” (или “`#`”), после которого в фигурных скобках указано тело события.

События и обработчики событий (или поля с соответствующими типами) связываются оператором “`<@`”.

Единственным оператором реактивного связывания является оператор “`<~`”, которым можно связывать, как отдельные поля, так и группы полей, объединенные фигурными скобками или привязывать поле (поля) к результату выполнения функции, посчитанной от других полей.

Графическая нотация

Графическая нотация языка Jarl во многом схожа с графическим языком технологии Room. Ниже представлен пример модуля Jarl.

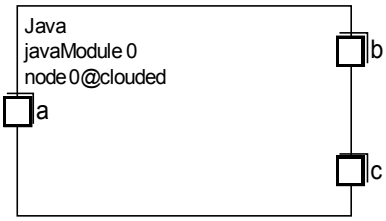


Рисунок 14: Пример модуля

На рисунке представлен сторонний модуль, который будет реализован на языке Java и у которого имеется три поля: a, b и c.

Теперь приведем пример обычного модуля.

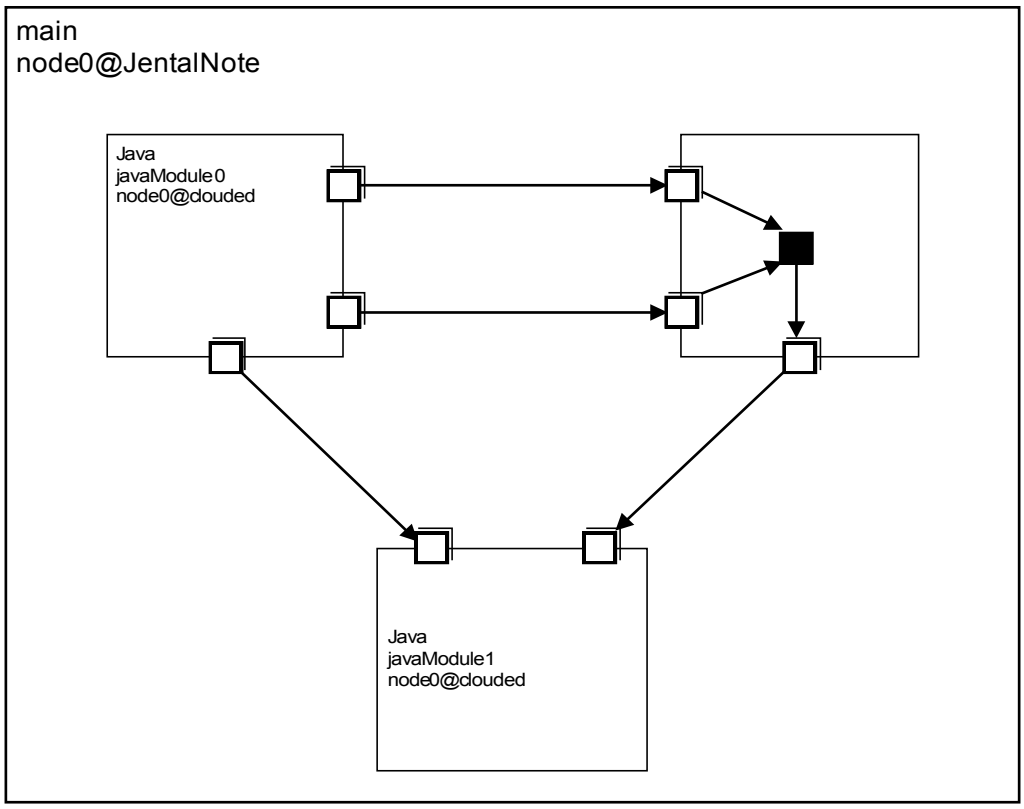


Рисунок 15: Пример модуля

Эта диаграмма определяет основной модуль, который будет работать на вычислительном узле node0@JentalNote и который содержит два сторонних модуля, написанных на языке Java и один модуль Jarl, который каким-то образом преобразует, используя локальное поле, два значения модуля javaModule0 и передает их второму модулю (javaModule1).

Глава 4. Реактивное программирование на языке Erlang

Одной из основных задач, которые потребовалось решить в процессе разработки технологии Jarl была организация реактивных связей между компонентами. Для ее решения был выбран язык Erlang – функциональный язык, предназначенный в первую очередь для написания распределенных систем и высоконагруженных серверных компонент.

В рамках данной задачи можно выделить две подзадачи: реализация состояний портов модулей в рамках функционального языка Erlang и организация реактивных связей между этими портами.

Поддержка состояний

Erlang, хоть и является функциональным языком, предоставляет некоторые возможности для хранения и изменения состояний.

- Словарь процесса — структура, уникальная для каждого процесса, в которой можно хранить пары ключ-значение.
- Распределенная база данных — мощное и достаточно тяжеловесное средство хранения данных.
- Общепринятое для функциональных языков хранение состояния в продолжении (continuation). Все состояния хранятся в некоторой структуре и передаются в качестве параметров следующему рекурсивному вызову функции, “слушающей” входящие сообщения.

В текущей реализации был выбран третий вариант. Но при этом, надо отметить, словари процессов тоже используются — в них хранятся данные, записанные при инициализации модуля и не изменяющиеся во время его работы. Такими данными, например, является идентификатор родительского (в терминах Jarl) процесса.

Таким образом основная функция процесса выглядит следующим образом:

```
listener(FieldToValueList) ->
  receive
    {_ConversationID, set, Sender, FieldName, {Type, Value}} ->
      listener(
        update_fields(FieldToValueList, {FieldName, {Type, Value}})
      )
      end;
    ...
    Other ->
      listener(
        FieldToValueList
      )
  end.
```

Внутри оператора “`receive … end`” располагаются шаблоны, с которыми сравниваются сообщения находящиеся в почтовом ящике (mailbox) процесса. Если какое-то сообщение совпадает с шаблоном, то вызывается соответствующий обработчик, указанный справа от оператора “->” соответствующего шаблона. Если ни одно из сообщений почтового ящика не совпадает с текущим шаблоном, то производится поиск следующего шаблона. Если сообщение не подходит ни под один шаблон, то процесс блокируется до получения следующего сообщения. В каждом из обработчиков вызывается функция-слушатель методом хвостовой рекурсии, что благодаря оптимизациям компилятора языка Erlang предотвращает возможность переполнения стека на этих вызовах.

На самом деле состояние, хранящееся процессом, несколько шире просто набора переменных со значениями. В процессе хранятся описанные ниже структуры.

- Значения полей, включающие в себя как собственно значения, так и типы полей.
- Ограничения доступа к полям для родительского модуля. Существует три “флага”, указывающие доступно ли поле для чтения, для записи и для

исполнения. Последний флаг нужен для возможности исполнения полей с типом “событий” или “обработчик”.

- Список привязок обработчиков к событиям, которые они обрабатывают.
- Список обработчиков конфликтных ситуаций. О них будет рассказано позже.

Библиотека для событийно-ориентированного программирования

Как и во многих других технологиях, обеспечивающих возможность реактивного связывания данных, реактивность в Jarl основана на событийно-ориентированной модели обработки данных. Поэтому в первую очередь в библиотеке была реализована именно поддержка событий.

Событие — триггер, инициируемый при изменении значения поля (или набора полей), к которым он привязан. Событие является одним из полей модуля, и его значение определяется в манере, схожей с определением дискретных событий в языке Yampa: событие — функция с двумя аргументами, возвращающая значение логического типа.

```
event :: ModuleState → Changes → bool
```

Первый аргумент – состояние полей модуля до инициации события,

второй – изменения полей, которые были применены к состоянию модуля,

результат — сработало ли событие или нет.

Обработчик события, как и событие является полем модуля и его значение — функция с двумя аргументами (теми же, что и у события), возвращающая список изменений, которые должны быть применены к состоянию модуля.

```
handler :: ModuleState → Changes → Changes
```

Поля-обработчики могут быть привязаны к одному или нескольким полям-событиям.

Ниже представлена диаграмма, показывающая принцип работы событийно-ориентированной модели, реализованный в библиотеке.

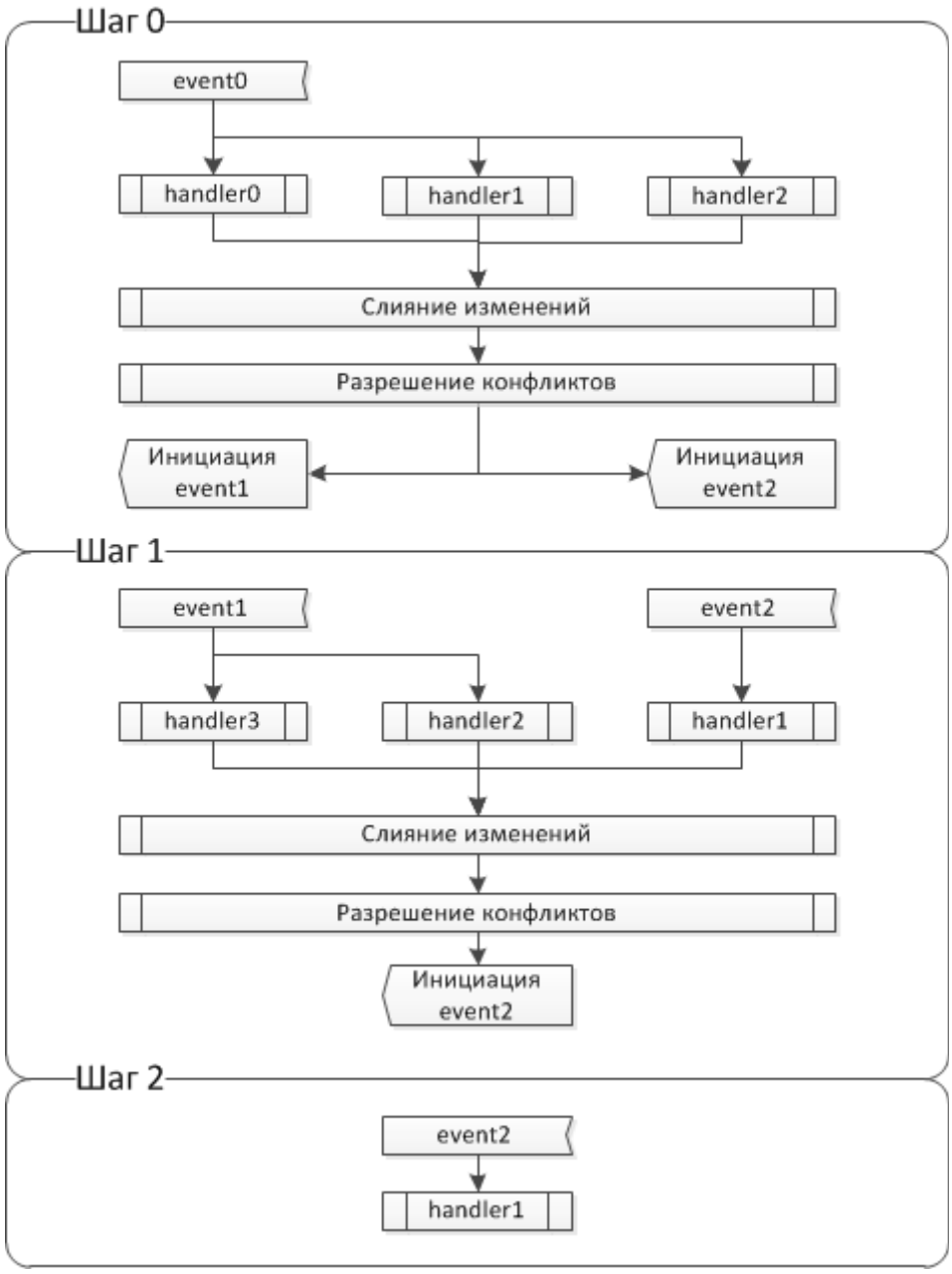


Рисунок 16: Событийный механизм

После того, как значение какой-либо порта (или несколько портов) модуля были изменены из родительского модуля, запускается каскадный механизм изменения состояния модуля. За этот механизм ответственна специальная функция-контроллер состояния модуля.

```
execute_front :: ModuleState → Changes → ModuleState
```

На вход эта функция получает два аргумента — текущее состояние модуля и список изменений полей, которые требуется применить, и возвращает новое состояние модуля.

В первую очередь функция-контроллер вычисляет фронт событий — события, которые сработали в соответствии с текущим состоянием и полученными изменениями. Затем для этого фронта вычисляется список обработчиков, которые требуется вызвать. Причем, поскольку всем этим обработчикам в качестве параметров будет передан один и тот же набор значений (текущее состояние модуля и полный набор изменений), список обработчиков на этом этапе можно сократить до списка, в котором каждый обработчик будет встречаться ровно один раз.

Затем все обработчики применяются, и в результате получается список, содержащий наборы изменений, которые требуется применить к состоянию модуля. Но дело в том, что среди этих изменений могут быть конфликты, т.е. разные обработчики могли назначить одному и тому же полю разные значения. Для разрешения таких ситуация были созданы обработчики конфликтов (resolvers).

Обработчик конфликтов — тип поля, со значением-функцией, которая из списка возможных значений поля выбирает одно, которое и будет новым значением этого поля.

```
resolver :: [FieldValue] → FieldValue
```

После применения обработчиков конфликтов, для каждого поля существует ровно одно значение, которое требуется применить. Теперь, если список новых изменений пуст, функция-контроллер возвращает текущее состояние системы, которое будет передано в качестве аргумента при следующем вызове слушателя. Если же список изменений непуст, то рекурсивно вызывается функция контроллер с обновленным состоянием модуля и новым списком изменений. Поля со значениями, являющимися обработчиками конфликтов, могут быть

привязаны к другим полям, и структура, содержащая эту информацию, также хранится в продолжении.

Такой механизм обработки событий блокирует получение сообщений модулем до конца своего исполнения (точнее сообщения будут получены и сложены в почтовый ящик процесса, но не будут обработаны). Это гарантирует то, что в процесс пересчета состояния модуля не сможет вмешаться какой-либо другой модуль. Но в то же время может возникнуть ситуация зацикливания в пересчете модуля. При этом, поскольку все вызовы функции-контроллера хвостовые, программа не завершится аварийно в связи с переполнением стека. Но новые сообщения не будут обработаны никогда. Для этого в следующей версии библиотеки планируется установка максимального числа фронтов событий, которые могут сработать при пересчете модуля, и реализация полей специального типа, которые могут бесконечно менять свои значений, не блокируя при этом обработку сообщений модулем.

Библиотека для реактивного программирования

Описанная в предыдущем разделе библиотека обладает исключительно интерфейсом на основе сообщений. Так, для изменения поля модуля, требуется послать ему сообщение, которое будет содержать имя поля, его новое значение, идентификатор сеанса общения и идентификатор процесса-оправителя, который при обработке сообщения будет сравнен с идентификатором родительского процесса или процессов-подмодулей, и, в случае положительного результата, присваивание будет выполнено. Очевидно, что такой механизм, в котором требуется посылать сообщение, содержащее идентификатор сеанса общения и идентификатор посылающего процесса неудобен для пользователя библиотеки. Намного удобнее было бы иметь набор функций, при помощи которого можно создать модуль, создать его поля и присваивать им значения, не заботясь о вещах, которые можно сгенерировать автоматически или получить из данных процесса. Для решения данной задачи была создана библиотека-обертка над основной библиотекой. И она, помимо удобного интерфейса для событийно-

ориентированного программирования, предоставляет возможности реактивного программирования на языке Erlang.

Для реализации корректной работы функций реактивного связывания в библиотеке-обертке существует набор вспомогательных шаблонов событий и их обработчиков. Так при помощи новой библиотеки можно легко создавать функцию-событие, которая будет реагировать на изменение только одного поля или нескольких полей, функцию-обработчик, которая изменит значение поля на текущее значение любого другого поля модуля или на какое-либо выражение, посчитанное на основе значений других полей.

```
event_OnChange :: FieldName → Event
event_OnChange :: [FieldName] → Event
handler_SetEqual :: FieldName → FieldName → Handler
handler_SetEqual :: FieldName → [FieldName] → ([Value] → Value) →
    Handler
```

Имея такие функции, довольно легко можно организовать механизм для создания реактивных связей между полями модулей. Требуется только создать событие на изменение поля (полей), находящегося в начале реактивной связи и привязать к нему обработчик, который изменит значение поля, находящегося в конце реактивной связи.

В связи с тем, что изменение локальных полей модуля происходит через механизм потока фронтов событий, а изменение сторонних полей — через сообщения, стоит различать реактивные связывания следующих типов:

- связывание локальных полей (включающее групповое связывание),
- связывание полей модуля и портов подмодуля.

Для них существуют соответствующие функции:

- `bind_sm2m(Process, Field, SubModuleName, Field_SM)`, реактивно связывающая порт подмодуля с полем модуля,
- `bind_m2sm(Process, Field, SubModuleName, Field_SM)`, делающая обратное действие,

- `bind_local(Process, Field_from, Field_to, Transformer, Sync)`, связывающая два локальных поля модуля с применением функции преобразования (существует и локальное связывание без преобразования),
- `bind_local_m(Process, Fields_from, Fields_to, Transformer, Sync)`, реактивно связывающая набор локальных полей с другим набором локальных полей с использованием функции преобразования.

Маршрутизатор

Функциональность приложения, описанная при помощи библиотеки, предназначена для исполнения в разных процессах, которые могут быть расположены на разных вычислительных узлах Erlang и на разных физических машинах. Для облегчения контроля за процессами и системой в целом было создано дополнение к библиотеке — маршрутизатор.

В языке Erlang существует функция `erlang::apply`, позволяющая исполнить функцию, переданную ей в качестве аргумента. По аналогии с ней (и частично MPI [40]) была создана функция, которая может выполнять функции библиотеки на удаленных вычислительных узлах Erlang.

```
exec :: Node → Module → Function → [Value] → unit
```

Тогда возможно не писать программы для каждого вычислительного узла отдельно, а выполнить следующие действия:

- написать одну программу, использующую механизм маршрутизатора,
- на каждом вычислительном узле запустить поток-слушатель exec-комманд,
- запустить программу на одном из вычислительных узлов.

Процесс работы с библиотекой

Поскольку большинство процессов в библиотеке (и ее обертке) выполняются асинхронно, для достижения предсказуемого результата следует придерживаться описанного ниже порядка исполнения.

1. Конструирование модулей, которое включает в себя следующие действия:
    1. собственно создание и инициализация модулей,
    2. создание полей модуля, обладающих значащими типами,
    3. создание полей-модмодулей,
    4. создание служебных полей (событий, обработчиков событий и обработчиков конфликтов) и их заполнение конкретными значениями
    5. локальное связывание полей модулей.

    Таким образом на этом этапе задается поведение модуля без указания конкретных значений для полей.

    В конце данного этапа рекомендуется вызвать функцию синхронизации wait для каждого создаваемого модуля.

    ```
    wait :: Process → unit
    ```

    Эта функция посылает сообщение каждому из модулей. Ее обработчик пуст, но он выполнится только после завершения всех остальных, что позволит получить по окончанию выполнения этой функции полностью сконфигурированный модуль, находящийся в стабильном состоянии.
2. Конфигурация модулей. На данном этапе выполняются:
    1. задание конкретных значений полям-подмодулям,
    2. реактивное связывание полей модуля и портов подмодуля.
3. Задание значений полям модуля.

Можно привести пример некорректной ситуации, появившейся в результате несоблюдения рекомендаций.

Пусть есть два модуля, причем один из них является подмодулем второго. Тогда, если поля первого модуля были инициализированы во время его создания (в том числе и события с обработчиками), может возникнуть ситуация, когда после изменения некоторого его поля, обработчик попытается изменить поле подмодуля, которое в этот момент может быть ещё не создано.

Глава 5. Генераторы кода

Генерация из текстовой нотации языка Jarl в Erlang

Итак, существует программа, написанная на языке текстовой нотации Jarl. Требуется получить для каждого Jarl-модуля модуль на языке Erlang.

При описании библиотеки, предназначенной для реактивного программирования на языке Erlang, описывался порядок действий по созданию и наполнению модулей, которому надо следовать для получения предсказуемо и правильно работающей программы. Очевидно, что сгенерированный код должен удовлетворять этому порядку. Поэтому для каждого поля генерируемого модуля, генерируется вхождения в три части программы, которые соответствуют трем этапам: созданию модуля, его конфигурированию, и заполнению модуля определенными значениями. После чего из отдельных фрагментов “собирается” целая программа, к которой в самом конце добавляются функции для запуска и инициализации модуля.

Для текстовой нотации языка была написана EBNF-грамматика [41], расширенная правилами построения абстрактного синтаксического дерева в удобной для генерации кода форме. По ней, при помощи разработанного на кафедре системного программирования СПбГУ инструмента YaccConstructor [5],[6], были сгенерированы лексический и синтаксический анализаторы на языке F# (.NET). На этом же языке и был написан генератор кода.

Генерация из текстовой нотации языка Jarl в Java

Для каждого стороннего модуля программы генерируется две сущности:

- некоторый интерфейс (шлюз) на целевом языке, предназначенный для взаимодействия в другими модулями,
- “обертка” над сторонним модулем, написанная на языке Erlang.

Вторая сущность (“обертка”) нужна по причине того, что целевые языки могут не поддерживать такие конструкции, как события и обработчики. Поэтому удобнее вынести всю функциональность такого рода в отдельный модуль, написанный на Erlang с использованием библиотеки для реактивного программирования. Порты этого модуля будут полностью дублировать порты стороннего модуля и будут связаны в ним на основе простого интерфейса, который будет дублировать изменившиеся значения портов.

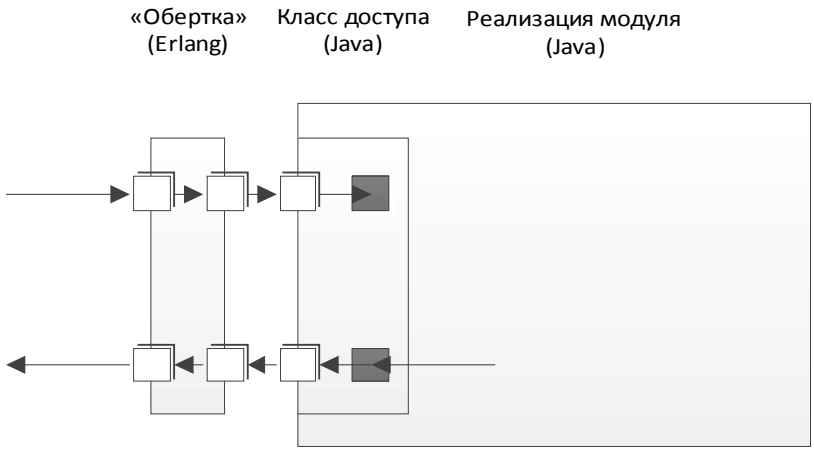


Рисунок 17: Сгенерированный код для стороннего модуля

“Обертка” является упрощенной версией Yarl-модуля и генерируется в Erlang по тому же принципу, который был описан в предыдущем разделе.

Шлюз в реализации для Java представляет из класс реализованный в соответствии с шаблоном “одиночка” (singleton), в котором каждому порту соответствует пара публичных методов, предназначенных для записи и чтения поля. Взаимодействие шлюза с “оберткой” осуществляется при помощи заранее написанной библиотеки, которая реализует следующую функциональность:

- создание, запуск и инициализация процесса,
- “общение” с “оберткой”,
- деинициализация процесса.

Библиотека написана при помощи библиотеки jinterface, предоставляющей интерфейс для взаимодействия с процессами Erlang/OTP для Java и официально поддерживаемой разработчиками языка Erlang.

Для генерации кода сторонних модулей был разработан механизм, предоставляющий возможность подключения плагинов к основному генератору. Плагин — динамическая библиотека среды .NET, лежащая с директории Plugins и содержащая хотя бы один тип, реализующий интерфейс IRLGeneratorPlugin, содержащий всего два метода: метод получения идентификатора языка (Language) и метод-генератор кода (Generate), получающий в качестве параметров абстрактное синтаксическое дерево, cookie и имя модуля. Такой механизм позволяет легко расширять набор языков, поддерживаемых технологией, причем генераторы могут быть написаны на любом языке платформы .NET.

Генерация из модели диаграммы QReal в текстовую нотацию языка Jarl

QReal – CASE-пакет, позволяющий при помощи относительно небольших усилий создавать новые графический языки и средства их обработки, в том числе и генераторы кода. Модель кода QReal обладает довольно сложно структурой, но это компенсируется удобным API для C++.

Для QReal был создан редактор, расширяющий его возможности для создания диаграмм в графической нотации языка Jarl.

Все элементы логической модели диаграммы в QReal образуют иерархию, по которой можно производить навигацию средствами предоставляемого API, при этом определяя тип текущего элемента. И, поскольку модуль, описанный в графической нотации языка Jarl, структурно совпадает с аналогичным, написанным в текстовой нотации, этого вполне достаточно для осуществления генерации кода.

Заключение

- Проведено исследование существующих технологий разработки распределенных и реактивных программ. Проведен обзор средств визуального моделирования систем.
- Разработаны две нотации языка Jarl
  - Текстовая
  - Графическая нотация

    Разработанная графическая нотация позволяет модулировать только системы, построенные на основе реактивных связей, но не событийно-ориентированные системы.
- Разработана библиотека, предоставляющая возможность событийно-ориентированного и реактивного программирования на языке Erlang. С помощью этой библиотеки осуществляются коммуникации между модулями.
- Разработаны генераторы кода для языков Erlang и Java.
- Разработан графический редактор графической нотации языка Jarl, реализованный как расширение функциональности CASE-пакета QReal.
- Разработан генератор, преобразующий диаграммы, созданные в приложении QReal, в код, написанный в текстовой нотации языка Jarl.
- Разработанный подход и описание языка были опубликованы на конференции СПИСОК-2011 [7].

Возможности развития

В дальнейшем предполагается следующие задачи.

- В настоящий момент генераторы и редакторы поддерживают только численные типы и операции над ними. Требуется расширение числа поддерживаемых типов символьным, строковым, логическим и структурными типами. Основной сложностью на данном этапе является корректная генерация преобразующих реактивных функций.
- Расширение графической нотации языка для поддержки событийно-ориентированного программирования.
- Разработка механизмов обработки ошибок модулей.
- Написание генераторов для других сторонних языков.

Список литературы

[1] Терехов А.Н., Романовский К.Ю., Кознов Д.В., Долгов П.С., Иванов А.Н., Real: Методолгия и CASE-средство для разработки систем реального времени и информационных систем // Программирование, 1999, № 5.

[2] Парфенов В.В., Терехов А.Н. RTST – технология программирования встроенных систем реального времени. // Системная информатика. Вып. 5: Архитектурные, формальные и программные модели. – Новосибирск, 1997, с. 228-256.

[3] Кознов Д.В. Визуальное моделирование компонентного программного обеспечения. Диссертация на соискание ученой степени кандидата физико-математических наук. Санкт-Петербургский Государственный Университет. 2000 г.

[4] А.Н. Терехов, Т.А. Брыксин, Ю.В. Литвинов и др., Архитектура среды визуального моделирования QReal. // Системное программирование. Вып. 4. СПб.: Изд-во СПбГУ. 2009, С. 171-196.

[5] Григорьев С.В. Генератор синтаксических анализаторов для неоднозначных контекстно-свободных грамматик. Дипломная работа. Санкт-Петербургский Государственный Университет. 2010 г.

[6] Улитин К.А., Кириленко Я.А. Модульный генератор синтаксических анализаторов // Технологии Microsoft в теории и практике программирования: материалы межвуз. конкурса-конф. студентов, аспирантов и молодых ученых Северо-Запада. 2011 / Российское представительство Microsoft; Санкт-Петербургский государственный политехнический университет; - СПб.: Изд-во Политехн.ун-та. 2011. С. 120.

[7] Соколов Н. Е. Подход к разработке распределенных гетерогенных реактивных систем / Соколов Н. Е., Луцив Д. В. // Материалы 2-й межвузовской

научной конференции по проблемам информатики «СПИСОК-2011». 2011. С. 18–22. http://spisok.math.spbu.ru/txt/SPISOK-2011.pdf (обр. Май-2012).

[8] McMurtry C., Mercuri M., Watling N. Microsoft Windows Communication Foundation: Hands-on! SAMS Publishing. 2006. 539 p.

[9] Henning M. A New Approach to Object-Oriented Middleware // IEEE Internet Computing. 2004 10 p.

[10] Alexy M., Korthaus A. Implementing Distributed Systems with Java and CORBA. Springer. 2005. 343 p.

[11] Simon St. Laurent, Johnston J., Dumbil E. Programming Web Services with XML-RPC. O'Reilly. 2001. 240 p.

[12] Kiczales, G.; Lamping, J; Mehdhekar, A; Maeda, C; Lopes, C. V.; Loingtier, J; Irwin, J. Aspect-Oriented Programming // Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241. June 1997

[13] Armstrong J. Making reliable distributed systems in the presence of software errors. PhD Thesis. The Royal Institute of Technology, Stockholm, Sweden. 295 p.

[14] Umut A. Acar. Self-Adjusting Computation. PhD Thesis. School of Computer Science, Carnegie Mellon University, Pittsburgh. 2005. 299 p.

[15] Petrícek T. Reactive programming with events. Master Thesis. Charles University, Prague. 2010. 147 p.

[16] Nilsson H. Dynamic Optimization for Functional Reactive Programming using Generalized Algebraic Data Types. // ACM. 2005. 12 p.

[17] Matthew A. Hammer, Umut A. Acar, Yan Chen CEAL: A C-Based Language for Self-Adjusting Computation Technical. Report TTIC-TR-2009-2, May 2009. 36 p. http://www.ttic.edu/technical_reports/ttic-tr-2009-2.pdf (valid. May 2012)

[18] Julien C. GruiaCatalin R. Active Coordination in Ad Hoc Networks // ACM, 2001 10 p.

[19] RFC 793. Internet protocol. Darpa internet program protocol specification. 1981. 45 p.

[20] RFC 793. Transmission control protocol. Darpa internet program protocol specification. 1981. 85 p.

[21] Formal/02-06-39 (CORBA 3.0 - OMG IDL Syntax and Semantics chapter) . 2002. 74 p.

[22] JSR-000220 Enterprise JavaBeans 3.0 Final Release (ejbcore). 2006. 562 p.

[23] Wang Yi-Min, Damani Om P., Lee Woei-Jyh . Reliability and availability issues in Distributed Component Object Model (DCOM). Position paper. 1997. 5 p.

[24] McLean S., Naftel J., Williams K. Microsoft .NET Remoting. Microsoft Press. 2002. 336 p.

[25] Bell M. SOA Modeling Patterns for Service-Oriented Discovery and Analysis. Wiley & Sons. 2010. 390 p.

[26] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. 2007. 103 p.

[27] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). 2007. 49 p.

[28] Giorgidze G., Nilsson H. Switched-on Yampa: Declarative programming of modular synthesizers. // Practical Aspects of Declarative Languages (PADL), 2008. Vol 4902 of Lecture Notes in Computer Science. Springer-Verlag. P. 282–298.

[29] Meyerovich L., Guha A., Baskin J., Cooper G., Greenberg M., Bromfield A., Krishnamurthi S. Flapjax: A Programming Language for Ajax Applications. Best Student Paper. OOPSLA. 2009. 20 p.

[30] Nordlander J. Reactive Objects and Functional Programming. PhD Thesis. Department of Computing Science Chalmers University of TechnologySE-412 96 Goteborg, Sweden. 1999. 208 p.

[31] Microsoft. Hands-on Lab Reactive Extensions for .NET, 2010. http://go.microsoft.com/fwlink/?LinkId=208528 (valid. May 2012).

[32] Microsoft. Hands-on Lab Reactive Extensions for Javascript, 2011. http://go.microsoft.com/fwlink/?LinkId=208527 (valid. May 2012).

[33] ITU Recommendation Z.100: Specification and Description Language. 1993. 204 p.

[34] ITU-T. Formal description techniques (FDT) – Specification and Description Language (SDL), 2002. http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf (valid. May 2012)

[35] Selic B., Gullekson G., Ward P.T. Real-Time Object-Oriented Modeling. John Wiley & Sons. Inc. 1994. 525 p.

[36] Paterson R. International Conference on Functional Programming. Firenze, Italy, 2001. P. 229-240.

[37] Jones S. P. Haskell 98 Language and Libraries: the Revised Report. Cambridge University Press. 2003. 272 p.

[38] Holt J. UML for Systems Engineering: Watching the Wheels. Institution of Electrical Engineers. 2004. 58 p.

[39] OMG. Formal/2006-01-01. Meta Object Facility (MOF) Core Specification. 2006. 88 p.

[40] Graham R.L., Shipman G.M., Barrett B.W., Castain R.H., Bosilca G., Lumsdaine A. Open MPI: A High-Performance, Heterogeneous MPI. Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks. 2006. 9 p.

[41] Roger S. Scowen: Extended BNF — A generic base standard. Software Engineering Standards Symposium.1993 10 p.