МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ Государственное БЮДЖЕТНОЕ образовательное учреждение

высшего профессионального образования

«Новосибирский НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ государственный университет» (нОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра параллельных вычислений

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

Пясс Инга Андреевна

Визуальный язык описания фрагментированных алгоритмов и транслятор с него в язык LuNA

Направление подготовки 230100.62 Информатика и вычислительная техника

**Руководитель Автор**

Киреев С.В. Пясс И.А.

(фамилия , И., О.) (фамилия , И., О.)

ФИТ, 9204

(уч.степень, уч.звание) (факультет, группа )

………….... ……………..

(подпись, дата) (подпись, дата)

Новосибирск, 2013 г.

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ Государственное БЮДЖЕТНОЕ образовательное учреждение

высшего профессионального образования

«Новосибирский НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ государственный университет» (нОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра параллельных вычислений

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.

(фамилия, И., О.)

………………….

(подпись, дата)

**ЗАДАНИЕ**

**НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

Студентке Пясс Инге Андреевне.

Направление подготовки 230100.62 Информатика и вычислительная техника

факультет информационных технологий

Тема: Визуальный язык описания фрагментированных алгоритмов и транслятор с него в язык LuNA.

Исходные данные (или цель работы): разработать визуальное представление фрагментированного алгоритма, реализовать визуальную среду разработки фрагментированных программ и реализовать транслятор для перевода графического представления алгоритма в представление на языке LuNA.

Структурные части работы: TODO

**Содержание**

[ВВЕДЕНИЕ 5](#_Toc357716240)

[Глава 1. Нужно придумать название 5](#_Toc357716241)

[1.1 Технология фрагментированного программирования 5](#_Toc357716242)

[1.2 Модель представления алгоритма 7](#_Toc357716243)

[1.2.1 Базовая модель представления алгоритма 7](#_Toc357716244)

[1.2.2 Модель алгоритма с условными операциями 7](#_Toc357716245)

[1.2.3 Иерархическая модель алгоритма 8](#_Toc357716246)

[1.3 Предварительная постановка задачи 8](#_Toc357716247)

[1.4 Обзор системы фрагментированного программирования LuNA 9](#_Toc357716248)

[Глава 2. Обзор систем визуального программирования 10](#_Toc357716249)

[2.1 Понятие визуального программирования 10](#_Toc357716250)

[2.2 Диаграмматические языки 11](#_Toc357716251)

[2.3 Иконические языки 14](#_Toc357716252)

[2.4 Итоги 15](#_Toc357716253)

[Глава 3. Реализация языка визуального программирования 16](#_Toc357716254)

[3.1 Средства для разработки 16](#_Toc357716255)

[3.1.1 Eclipse Modeling Framework 17](#_Toc357716256)

[3.1.2 Graphical Editing Framework 17](#_Toc357716257)

[3.2 Разработка и представление модели 17](#_Toc357716258)

[3.2.1 Visual LuNA overview 17](#_Toc357716259)

[3.2.2 Model overview 21](#_Toc357716260)

[3.2.3 Eclipse GMF overview 22](#_Toc357716261)

[3.2.4 Транслятор из Visual LuNA в LuNA 28](#_Toc357716262)

[ЗАКЛЮЧЕНИЕ 30](#_Toc357716263)

[Список использованных источников 31](#_Toc357716264)

# ВВЕДЕНИЕ

**Проблема параллельного программирования**

Разработка хорошей параллельной программы в настоящее время остаётся сложной задачей. Параллельная программа должна обладать рядом динамических свойств (динамической настройкой на доступные ресурсы, динамической балансировкой загрузки, динамическим учётом поведения численной модели), которые существующими системами программирования в полной мере не обеспечиваются, а их ручная реализация требует серьёзных знаний теории параллельного программирования, архитектуры ЭВМ, программных особенностей средств разработки. В параллельных программах встречаются специфические ошибки, которые сложно обнаружить и устранить в силу недетерминизма исполнения таких программ. Наконец, перенос параллельной программы с одного вычислителя на другой часто также сопряжён со значительными трудностями.

Таким образом, специалистам-прикладникам приходится уделять слишком большое внимание проблемам параллельного программирования. В связи с этим, одним из перспективных направлений является создание систем параллельного программирования высокого уровня, которые в значительной степени скрывают от пользователя технические трудности разработки параллельных программ и позволяют сосредоточиться на алгоритме решения задачи.

Одним из новых подходов, альтернативных программированию на основе параллельно-последовательной модели вычислений, является фрагментированное программирование. К его основным преимуществам можно отнести возможность автоматической генерации параллельной программы, возможность автоматически обеспечить программам ряд динамических свойств, лучшую переносимость между вычислителями различной архитектуры, а также возможность выполнить формальный анализ параллельной программы. Этот подход в настоящее время активно развивается, и в существующих проектах (PLASMA, ALF, Charm++) реализованы лишь его отдельные элементы. Поэтому дальнейшее развитие средств фрагментированного программирования является актуальной задачей.

# Глава 1. Нужно придумать название

## 1.1 Технология фрагментированного программирования

Итак, давайте разберёмся, что же представляет собой фрагментированное программирование. Мы уже перечислили основные преимущества данного подхода:

* возможность автоматической генерации параллельной программы;
* хорошая переносимость между вычислителями различной архитектуры;
* возможность выполнить формальный анализ параллельной программы.

В чём же суть данного подхода и за счёт чего вышеперечисленные преимущества достигаются?

Любой алгоритм можно представить в виде множества фрагментов данных и фрагментов кода. Фрагмент кода получает на вход набор входных фрагментов данных, на основе которых вычисляет набор выходных фрагментов данных. Совокупность фрагмента кода и его входных и выходных фрагментов данных называется фрагментом вычислений. На множестве фрагментов вычислений задаётся строгий частичный порядок.

Фрагментированный алгоритм содержит только минимальное управление, определяемое зависимостями по данным, и не содержит распределения ресурсов. Таким образом, он допускает множество способов исполнения, что обеспечивает его переносимость. Задача исполнительной системы – выполнить отображение объектов алгоритма (переменных, операций) на ресурсы конкретной вычислительной системы, автоматически обеспечивая все необходимые динамические свойства параллельной программы. Фрагментация – это технологический прием, позволяющий уменьшить число объектов алгоритма и тем самым упростить задачу построения эффективного распределения ресурсов и управления.

Таким образом, можно придти к выводу, что фрагментированная программа может быть представлена в виде ориентированного ациклического графа:

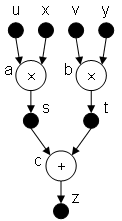
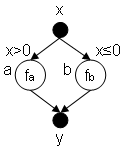
 

Рисунок 1 - Примеры простых алгоритмов. Слева - алгоритм вычисления выражения z = ux + vy. Справа - алгоритм с условным оператором для вычисления выражения if x > 0 then y = x else y = -x

Модель фрагментированной программы достаточно простая, а визуальное программирование в потенциале более емкое и наглядное, чем обычное, использующее линейное текстовое представление, поэтому постоянно делаются попытки создания визуальных языков программирования. Исходя из вышеописанного, становится очевидной актуальность разработки графического представления фрагментированной программы. Это позволит сделать процесс разработки программы ещё более наглядным и понятным.

## 1.2 Модель представления алгоритма

* Требования к модели:простая.
* Cохраняет параллелизм алгоритма.
* Позволяет эффективно реализовать алгоритмы на параллельных вычислителях.

### 1.2.1 Базовая модель представления алгоритма

Алгоритм – это четверка , где:

* – множество переменных,
* – множество операций,
* : – отношение «вход»,
* : отношение «выход».

Алгоритм – двудольный ориентированный ациклический граф с вершинами двух видов: переменные единственного присваивания и операции единственного срабатывания (тождественно определению вычислимой функции Клини). Дуги графа обозначают отношения «вход-выход».

Каждая переменная модели может находиться в двух состояниях: означенном (ей сопоставлено конкретное значение) и неозначенном (конкретное значение не сопоставлено). Перед началом работы алгоритма некоторые переменные получают значения. В процессе работы алгоритма каждая операция может сработать независимо от других, если все ее входные переменные получили значения. В результате срабатывания операции значения получают ее выходные переменные. Исполнение завершается, если ни одна операция больше не может сработать. Чтобы избежать неоднозначности при означивании переменных, введем следующее «правило уникальности»: только одна операция может иметь какую-либо переменную в качестве выходной.

### 1.2.2 Модель алгоритма с условными операциями

Чтобы представлять алгоритмы с ветвлениями, требуется ввести в модель понятие условной операции. Условная операция – это операция, которой сопоставлено предусловие, т.е. некоторое логическое выражение, вычисляемое в зависимости от значений входных переменных этой операции. Условная операция может быть исполнена, если значение ее предусловия равно «истина». Если значение предусловия оказалось «ложь», то условная операция удаляется из алгоритма. Вместе с ней удаляются все переменные, которые могли быть означены только при условии ее выполнения, и все операции, которые могли исполниться только при условии означивания удаляемых переменных. Обычные операции рассматриваются как условные с предусловием, тождественно равным «истина».

С введением условных операций «правило уникальности» должно быть соответствующим образом модифицировано: если несколько условных операций имеют на выходе одну и ту же переменную, то области истинности их предусловий не должны пересекаться.

### 1.2.3 Иерархическая модель алгоритма

Введем в модель представления алгоритма понятие агрегации. Позволим некоторое множество переменных представлять как одну агрегированную переменную, а некоторое множество операций – как одну агрегированную операцию. При этом отношения «вход-выход» также соответствующим образом агрегируются.

Кроме того, введем понятие структурированной операции (соответствующей понятию подпрограммы в процедурных языках программирования), которая агрегирует в себе как переменные, так и операции, включая. На агрегацию переменных и операций в структурированную операцию накладывается следующее естественное ограничение: если между агрегируемыми объектами есть путь в графе алгоритма (информационная зависимость), то этот путь должен также содержаться в структурированной операции.

## 1.3 Предварительная постановка задачи

Объектами исследования являются:

* фрагментированное программирование,
* визуальное программирование,
* язык LuNA.

Целью исследования является:

* Разработать визуальное представление фрагментированного алгоритма.
* Реализовать визуальную среду разработки фрагментированных программ.
* Реализовать транслятор для перевода графического представления алгоритма в представление на языке LuNA.

## 1.4 Обзор системы фрагментированного программирования LuNA

Для применения технологии фрагментированного программирования разрабатывается экспериментальная система фрагментированного программирования LuNA. Сейчас она включает следующие компоненты:

* язык фрагментированного программирования LuNA,
* компилятор,
* планировщик,
* runtime-система.

Язык LuNA позволяет описать фрагментированную программу как множество фрагментов, а также содержит средства для прямых указаний и рекомендаций по распределению ресурсов. Задача планировщика – частичное статическое планирование вычислений на основе имеющейся информации о программе, о вычислительной системе и с учетом указаний и рекомендаций пользователя. Runtime-система обеспечивает исполнение фрагментированной программы на имеющемся оборудовании. В ходе своей работы она должна выполнять динамическое распределение ресурсов, автоматически обеспечивая программе необходимые динамические свойства: автоматическую настройку программы на имеющиеся ресурсы, динамическую балансировку загрузки, организацию асинхронных обменов данными и вычислений.

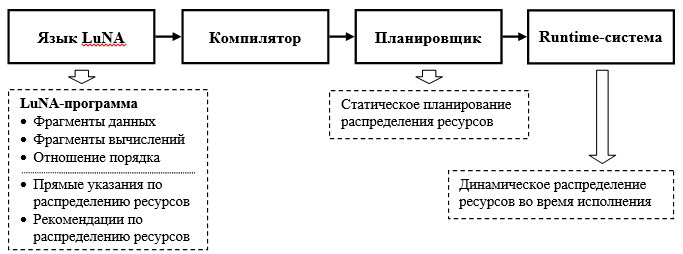


Рисунок 2 - Схема работы системы фрагментированного программирования LuNA

В системе LuNA, фрагментированная программа представляет собой ориентированный ациклический граф. Основное отличие от других подобных систем параллельного программирования заключается в наличии рекомендаций, которые дают возможность более гибкого управления распределением ресурсов. В качестве дополнительной информации пользователь может задать следующие сведения о программе:

* приоритеты фрагментов вычислений,
* отношение соседства между фрагментами вычислений,
* объединение фрагментов вычислений в группы, чтобы при запуске одного фрагмента из группы, повышался приоритет остальных фрагментов группы,
* веса фрагментов вычислений, дающие оценку времени их работы.

Также пользователь может напрямую указать распределение фрагментов вычислений по узлам и ужесточить порядок исполнения, добавив новые зависимости.

Вывод: так как программа в системе LuNA представляет собой ориентированный ациклический граф, то мы можем разработать визуальное представление для этой системы. Теперь нужно разработать детали концепции визуального языка фрагментированного программирования.

# Глава 2. Обзор систем визуального программирования

## 2.1 Понятие визуального программирования

Визуальное программирование – попытка упростить процесс программирования за счет большей емкости и наглядности представления программы, что позволяет более эффективно использовать возможности человеческого разума[1]. Отличия подхода визуального программирования от обычного текстового программирования:

* Изображения являются более мощным инструментом нежели чем слова. Они могут передать больше смысла в более краткие единицы выражения.
* Изображения проще понять и запомнить.
* У изображений нет языковых барьеров.

Не существует единого формального определения языка визуального программирования. Следующее определение даёт Nan.C.Shu в книге Visual Programming (1988)[2]: *“The use of meaningful graphic representation in the process of programming.”* (т.е. “использование значимого графического представления в процессе программирования”).

Языки визуального программирования можно классифицировать по двум параметрам. Первый параметр, по которому можно классифицировать – парадигмы (как и любой другой язык программирования), например, логический, объектно-ориентированный и т.п. Второй – по визуальному представлению, по которому языки визуального программирования можно разделить на 3 категории[3]:

* Diagrammatic languages (диаграмматические языки)
* Iconic languages (иконические языки)
* Languages based on static pictorial sequences.

Самые распространенные типы – диаграмматические и иконические. Их мы в данном обзоре и рассмотрим.

## 2.2 Диаграмматические языки

Диаграмматические языки[4] основаны на использовании схем и диаграмм, описывающих программные процессы. Они характеризуются строго определенным, формализованным словарем, состоящим из сравнительно небольшого количества элементов. Также, как правило, строго определен пространственный синтаксис – расположение элементов схем в пространстве и относительно друг друга. Многие системы проводили размещение элементов схем автоматически. Работа с системами программирования на базе диаграмматических языков, как правило, идет по общей схеме – пользователь, применяя систему меню, указывает на графические шаблоны – элементы диаграмм и размещает их в рабочем поле. После этого происходит заполнение соответствующих текстовых полей шаблонов.

Таким образом, диаграмматические языки программирования являются гибридными языками, т.е. используют не только графические элементы для создания программ, но и текстовые.

Примером диаграмматических языков является Microsoft Visual Programming Language[5].Язык программирования Microsoft Visual Programming Language (VPL) — это средство разработки приложений, в котором используется модель программирования на основе графического потока данных, а не на базе логики управления, которая обычно применяется в программировании. Программа с использованием потока данных — это не набор последовательно выполняемых обязательных команд. Она больше напоминает коллектив рабочих на сборочной линии, которые делают указанную им работу по мере поступления деталей. В результате язык VPL хорошо подходит для программирования различных сценариев с параллельной или распределенной обработкой данных.

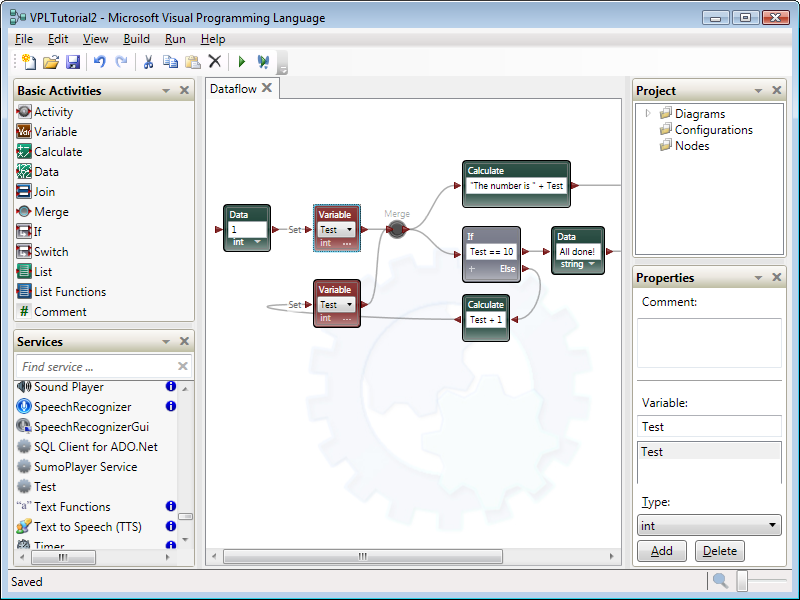


Рисунок 3 - Пример диаграммы MVPL

Поток данных MVPL состоит из связанной последовательности так называемых activities, представленных в виде блоков с входами и выходами, при помощи которых этот блок можно присоединить к другим activity-блокам.

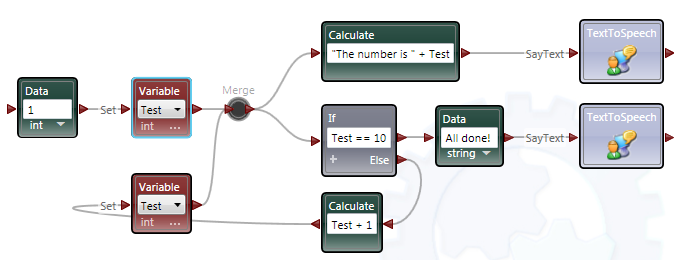


Рисунок 4 - Activity-блоки имеют соединения, которые представляют собой сообщения, отправленные с одного activity на другой

Activities могут представлять управление потоком данных или обработкой функций. Соединения между activities обозначают передачу данных в виде сообщений.

Activities также могут включать в себя другие activities. Это делает возможным создание своих собственных activities, путём комбинирования существующих, а затем использовать композицию в качестве строительных блоков. В этом смысле приложение, созданное в VPL само по себе является activity.

Activity-блок обычно включает название activity и точки соединения. Activity-блок также может включать в себя графику для иллюстрации цели activity или элементы пользовательского интерфейса, такие как текстовые поля, которые позволяют пользователю вводить значения, присваивать или изменять данные, используемые в activity.

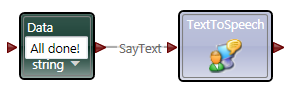


Рисунок 5 - Подключения вызывают действия (в данном случае SayText)

Activity могут иметь множественные входы для подключения, каждый со своим собственным набором выходов. Входы представлены стрелками, входящими в блок. Выход может быть одним из двух типов: result (ответ)  или notification (иногда называется event или publication). Результирующие выходы представлены стрелками, выходящими из блока, выход-уведомление представлен в виде круга:

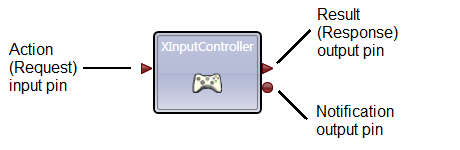


Рисунок 6 - Connection pins

## 2.3 Иконические языки

Иконические языки[4] используют иконы (пиктограммы) для представления объектов, операций и функций. Так как достаточно сложно передавать всю информацию через изображения, иконические языки сильно уступают диаграмматическим. Иконические языки ни для чего серьезного не используются и как правило применяются для обучения детей программированию.

Примером современного иконического визуального языка программирования является Kodu[6].Это новый язык программирования для создания игр. Он разработан быть доступным для детей и интересным для всех возрастов. Среда программирования запускается на игровой платформе Xbox и управляется джойстиком.

Сердце проекта Kodu — программируемый пользовательский интерфейс. Язык простой и полностью основан на иконках. Программы состоят из страниц, которые разбиваются на правила, которые в свою очередь разделяются на условия и действия. Условия высчитываются одновременно.

Язык Kodu разработан специально для разработки игр и предоставляет специализированные примитивы, полученные из игровых сценариев. Хотя Kodu не является языком общего назначения, как классические языки программирования, Kodu может предоставить концепции игрового дизайна в простой и интуитивно-понятной манере.

Для разработки игры необходимо создать игровой мир, в котором будут жить внедрённые программистом персонажи, и взаимодействовать по установленным им правилам (а также с учётом законов физики). При начальном запуске игры можно загрузить множество существующих миров, или же начать с пустого мира.

Внутри мира живут объекты («коду» - персонажи (нечто вроде животного), деревья, пушки, яблоки и т.п.). У объектов есть поведение – например, пушка, добавленная в игровой мир, может вдруг сама по себе опрокидываться на спину и «улыбаться». Программист может сам задавать поведение – например при нажатии на «F1» c дерева упадет яблоко.

Простой пример программы – создаем пустой мир, добавляем в него землю, на нее дерево и подводную лодку. Подводной лодке задаем свойство, что у нее есть пушка. Добавляем пушке снаряды, которым также можно задать цвет. А дальше начинается самое интересное – программирование поведения. Например, при нажатии кнопки влево выбираем «повернуться», затем выбираем куда повернуться – «влево». При нажатии кнопки вправо, выбираем что объекту нужно «повернуться», потом выбираем, что повернуться нужно «вправо». При нажатии на кнопку «пробел» зададим поведение «выстрелить», зададим чем выстрелить – каким снарядом, зададим цвет снаряда случайно.

Таким образом, чтобы создать игру – нужно создать мир, создать внутри него какие-то объекты и наделить их поведением. В ответ на нажатия клавиш, мышки можно заставить объекты перемещаться, улыбаться, переходить в новое состояние

Пара скриншотов, отражающих суть создания игры:



Рисунок 7 - добавление объекта в игровой мир

****

Рисунок 8 - задание свойств объекта

****

Рисунок 9 - задание поведения объекта

## 2.4 Итоги

Рассматривая системы визуального программирования, можно придти к выводу, что диаграмматические языки лучше подходят для представления фрагментированных алгоритмов.

Несмотря на все преимущества, у визуального подхода к программированию есть ряд проблем, которые нужно учесть при разработке графического представления алгоритма:

* Громоздкое изображение — не помещается на экран.
* Ограниченность языка – при любом экране объем визуальных данных, необходимых для представления реальной параллельной программы, превосходит его возможности и возможности восприятия программистом.
* Возможны сложные визуальные тексты - даже не очень большое усложнение структуры программы приводит к запутанным картинкам, напоминающим по сложности интерпретации пазлы.

**2.5 Выводы**

В главах 1 и 2 мы рассмотрели преимущества и недостатки:

* фрагментированного программирования,
* существующих систем визуального программирования,
* системы фрагментированного программирования LuNA.

На основании всего вышеописанного можно сделать выводы, какой должна быть наша модель визуального программирования, возможные проблемы при реализации и подходы для решения этих проблем.

* Язык визуального программирования должен быть диаграмматическим.
* В палитре инструментов должны быть фрагменты данных, фрагменты вычислений, связи между ними и операторы визуального языка.
* Для решения проблемы многих ныне существующих языков визуального программирования – громоздких изображений, будет использован подход, в котором каждый структурированный фрагмент вычислений будет описываться отдельной диаграммой.

# Глава 3. Реализация языка визуального программирования

## 3.1 Средства для разработки

В качестве средства для разработки языка визуального программирования была выбрана платформа Eclipse. Eclipse предоставляет два фреймворка для реализации систем визуального программирования:

* Eclipse Modeling Framework
* Graphical Editing Framework

### 3.1.1 Eclipse Modeling Framework

Разработка любого приложения начинается с разработки модели приложения, а затем переходит в реализацию отдельных его компонентов, в том числе интерфейса. Eclipse Modeling Framework предназначен для облегчения процесса разработки и реализации модели. Этот фреймворк позволяет сосредоточиться на модели приложения, а не на деталях реализации. По модели генерируется Java-код, который в дальнейшем может использоваться в реализации приложения.

### 3.1.2 Graphical Editing Framework

Graphical Editing Framework позволяет легко разрабатывать графические представления для существующих моделей. Редактор, сгенерированный при помощи Graphical Editing Framework состоит из следующих компонентов:

* Палитра инструментов редактора
* Изменение свойств существующих элементов
* Drag&Drop, Undo-Redo, меню, панели инструментов.

Graphical Editing Framework и Eclipse Modeling Framework представляют собой Graphical Modeling Framework[7,8], который и был использован мной при написании данной работы.

## 3.2 Разработка и представление модели Visual LuNA

### 3.2.1 Графический язык представления алгоритмов

В пункте 1.2.1 была описана базовая модель алгоритма. Алгоритм состоит из множества переменных, множества операций и отношений «вход», «выход». На основе этой четверки, можно задать особые типы компонентов:

* условные операторы (пример представления алгоритма с условным оператором был представлен на рисунке 1);
* операторы for и while

aN

xN

yN

zN

s

b

a1

a2

x1

x2

y1

y2

z1

z2

…

…

…

s

b

ai

xi

yi

zi

x

y

i = 1…N

uy

ux

N

Рисунок 10 - пример алгоритма с массивом (for i =1..N)

* встроенные фрагменты вычислений (копирование значения, или присваивание)

Графическое представление компонентов:

1. Фрагмент вычислений представлен синим эллипсом

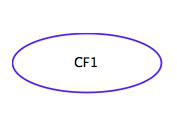


Рисунок 11 - представление фрагмента вычисления

1. Фрагмент данных представлен розовым закругленным прямоугольником



Рисунок 12 - представление фрагмента данных

1. Встроенный фрагмент вычислений – присваивание представлен синим кругом (именно потому что он является фрагментом вычислений)

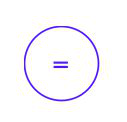


Рисунок 13 - представление присваивания

1. Операторы for, while, if и выражение представлены серым прямоугольником и различаются только параметрами и соответсвующей подписью сверху прямоугольника.

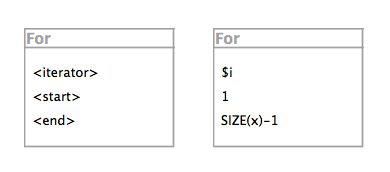


Рисунок 14 - представление оператора for

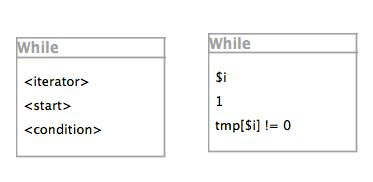


Рисунок 15 - представление оператора while

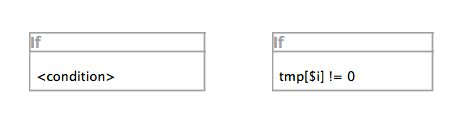


Рисунок 16 - представление условного оператора if



Рисунок 17 - представление выражения

1. Комментарий представлен ярко-желтым прямоугольником.

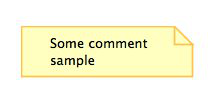


Рисунок 18 - представление комментария

1. На дугах (стрелках, связях) указывается имя входного в фрагмент вычислений аргумента, а также индекс соответствующего фрагмента данных.

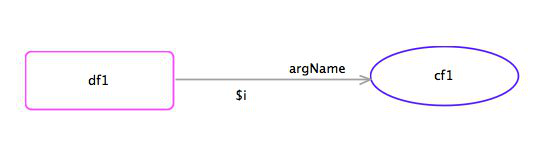


Рисунок 19 - представление связей (отношения «вход», «выход»)

### 3.2.2 Visual LuNA overview

С использованием Graphical Modeling Framework на платформе Eclipse была разработана и реализована среда визуального программирования Visual LuNA. На рисунке 8 представлен внешний вид среды. Основными компонентами среды являются (обозначены соответствующими цифрами на рисунке):

* 1. проекты, содержащие фрагменты кода и представляющие собой фрагментированную программу
  2. палитра инструментов,
  3. область редактирования алгоритма (канвас).

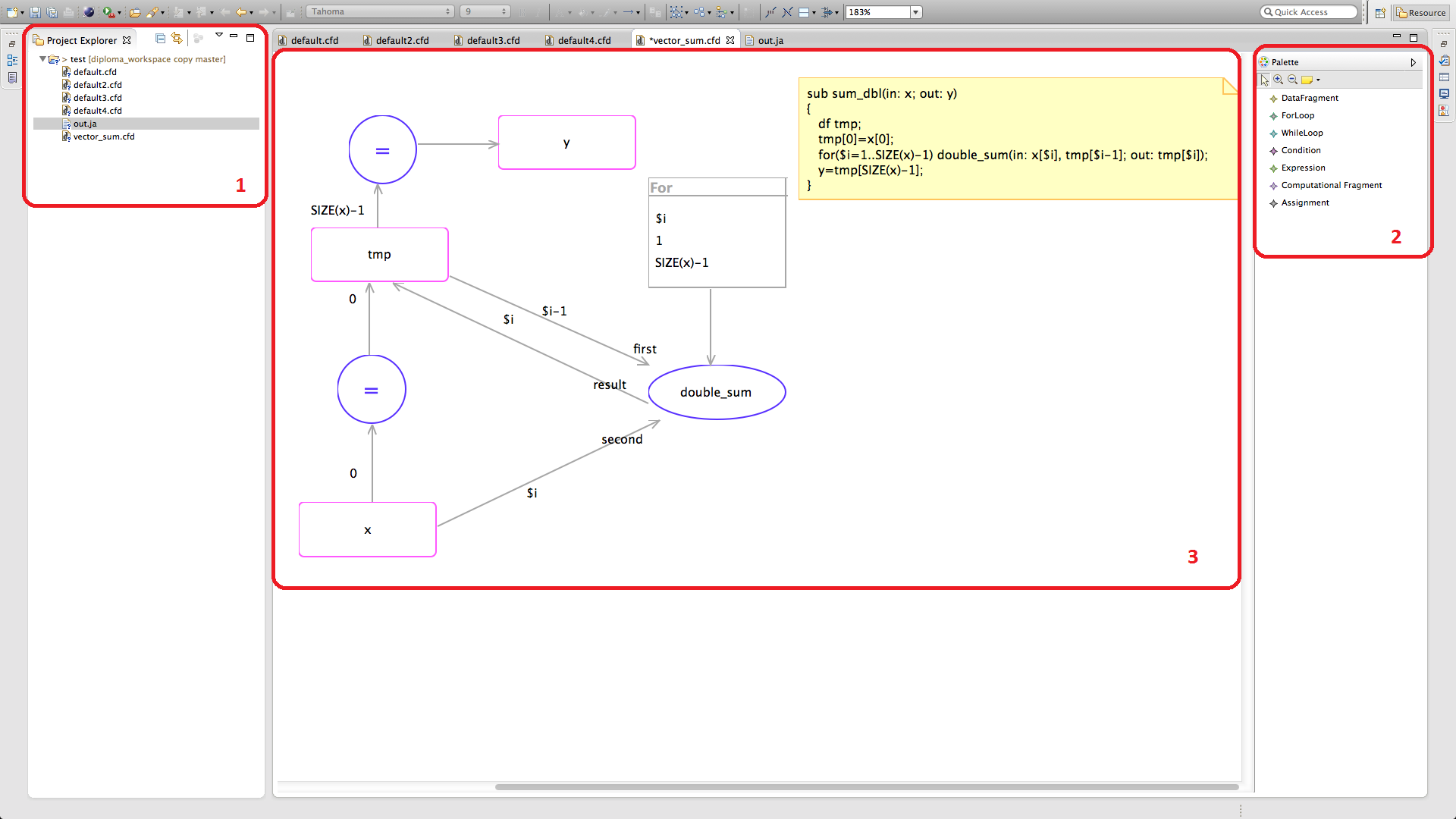


Рисунок 20 - Внешний вид среды визуального программирования Visual LuNA

Палитра инструментов располагается справа. На палитре инструментов расположены следующие инструменты:

1. DataFragment (фрагмент данных)
2. ForLoop (оператор for порождения множества)
3. WhileLoop (оператор while порождения множества)
4. Condition (условный оператор if)
5. Expression (арифметическое выражение, например SIZE(x) = 5)
6. Computational Fragment (фрагмент вычислений)
7. Assignment (присваивание, является встроенным фрагментом вычислений)

На палитре инструментов также предусмотрен инструмент «комментарий», представляющий собой поле для ввода текста комментария, и «масштаб», предоставляющий возможность увеличивать или уменьшать масштаб канваса.

Канвас представляет собой место отображения фрагмента кода. На канвас можно добавлять инструменты из палитры инструментов и добавлять связи между ними (при помощи перетягивания стрелок, которые появляются когда наводишь на инструмент). На канвасе можно менять свойства объектов (например, название, имя аргумента, индекс и т.п.).

Теперь, когда мы немного разобрались с визуальным представлением графического редактора можно приступить к рассмотрению модели Visual LuNA.

### 3.2.3 Model overview

Модель Visual LuNA представлена на следующем рисунке:

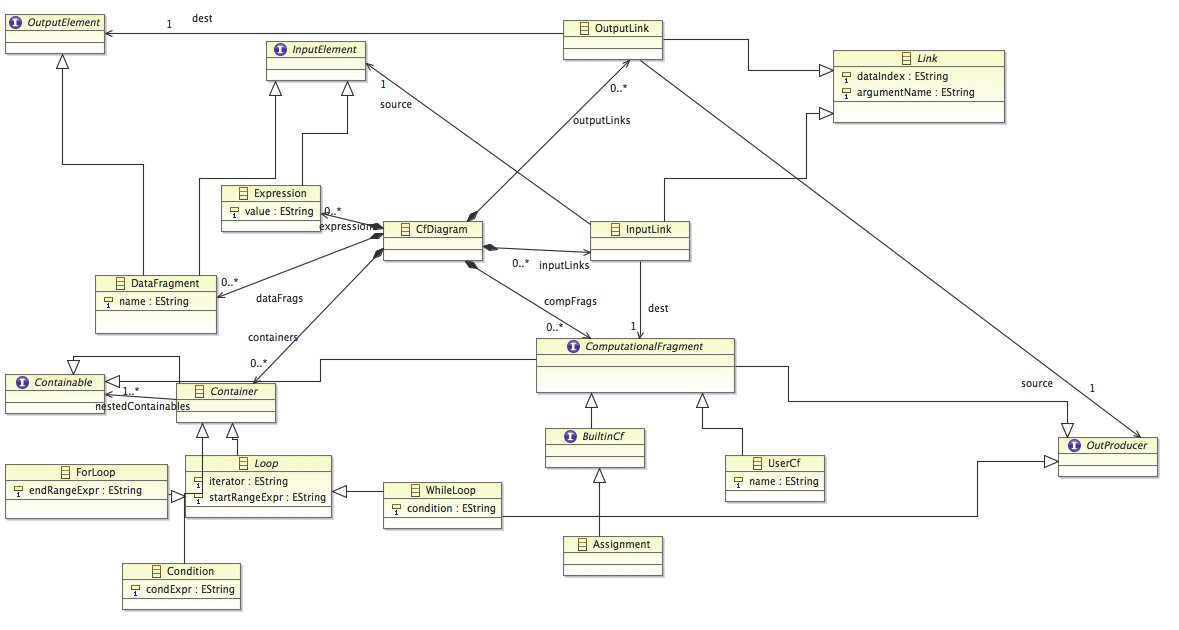


Рисунок 21 - модель Visual LuNA в представлении Eclipse Modeling Framework

Главным классом является класс *CfDiagram*, который по сути представляет нашу диаграмму (фрагмент кода). Данный класс содержит в себе следующие классы:

1. *DataFragment* – фрагмент данных, который является наследником двух интерфейсов – *InputElement* и *OutputElement* (соответственно входной и выходной фрагмент данных).
2. *ComputationalFragment* (интерфейс) – фрагмент вычислений. В модели было сделано два вида фрагментов вычислений: встроенные – представлены интерфейсом *BuiltinCf* и те, которые программист описывает диаграммами – представлены интерфейсом *UserCf*. Встроенный фрагмент вычислений на данный момент один – присваивание (представлен наследником класса *BuiltinCf* – *Assignment)*, но это позволяет при необходимости расширить и дополнить модель. Фрагмент вычислений описанный программистом (класс *UserCF*) имеет одно поле – название данного фрагмента вычислений, для того чтобы потом можно было ссылаться на него в других фрагментах кода.
3. *InputLink, OutputLink* – входящая/выходящая дуга (первая входит во фрагмент вычислений из объекта InputElement, вторая входит во фрагмент данных из объекта OutProducer). Оба класса наследуются от общего класса *Link* содержащего два поля: *dataIndex* (индекс фрагмента данных)и *argumentName* (имя входного аргумента). Для заданного ФВ (объекта класса ComputationalFragment) набор его входных аргументов можно получить, отсеяв все объекты класса *InputLink*, поле *dest* (назначение) которых не соответствует заданному ФВ. Далее, объединив значения полей *source* (источник) для полученных дуг, мы получим множество входных аргументов для заданного ФВ. Набор выходных аргументов может быть получен путём проделывания аналогичных операций для объектов класса *OutputLink*.
4. *Expression* – выражение, например . Имеет одно поле – *value* (как раз в котором указано выражение).
5. *Container* – Символизирует некоторый блок (контейнер), который может содержать имплементации интерфейса *Containable*, которые могут представлять собой фрагменты вычислений либо другие блоки. Контейнерами в модели являются операторы for и while (наследники класса *Loop*) и условный оператор (класс *If*). Абстрактный класс Loop содержит атрибуты, общие для операторов for и while, — это имя счетчика (поле *iterator*) и его начальное значение (*startRangeExpr*), представляющее собой выражение. *ForLoop —* наследник класса Loop, представляющий собой оператор for. Он расширяет своего предка наличием заранее известного конечного значения счетчика (*endRangeExpr*). *WhileLoop* — наследник класса Loop, представляющий оператор while. Для оператора while конечное значение счетчика неизвестно, однако присутствует условное выражение (*condition*), определяющее необходимость продолжения итераций. В модели ФП конечное значение счетчика записывается во фрагмент данных, который является «выходным» для оператора while. Именно поэтому WhileLoop является имплементацией интерфейса OutProducer. Класс *If* представляет собой условный оператор. Содержимое этого блока будет исполняться только если условное выражение (*condExpr*) этого оператора истинно.
6. *OutProducer –* интерфейс, символизирующий объект, способный порождать выходные значения (OutputElement). Он является источником (source) для объектов класса OutputLink. Этот интерфейс имплементируют абстракции ФВ и оператора while.

### 3.2.4 Eclipse GMF overview

Как уже говорилось ранее – Graphical Modeling Framework состоит из двух фреймворков – Eclipse Modeling Framework и Graphical Editing Framework. Теперь можно рассмотреть из каких компонент состоит редактор в представлени этих двух фреймворков.

На следующем изображении представлена панель Graphical Modeling Framework:

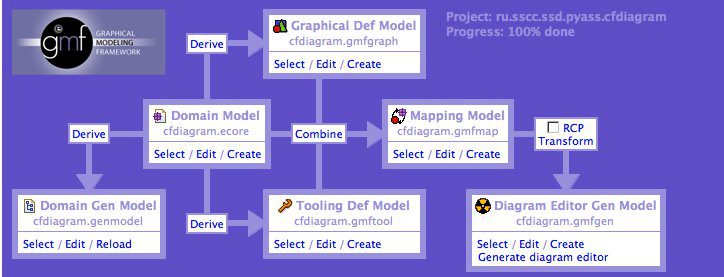


Рисунок 22 - панель GMF

GMF использует шесть файлов для создания графического редактора, как показано на скриншоте выше. Первые два файла – cfdiagram.ecore и cfdiagram.genmodel относятся к фремворку EMF, который используется для описания модели. В EMF есть два понятия – мета-модель и модель. Мета-модель описывает структуру модели. EMF содержит информацию о модели в формате XMI (XML Metadata Interchange). Eclipse Modeling Framework позволяет по-разному описать мета-модель – при помощи UML-диаграммы или XML-схемы. При описании мета-модели при помощи UML-диаграммы по ней генерируется XML-схема.

Ключевым сердцем данной модели редактора является мета-модель – cfdiagram.ecore. Она представлена в виде cfdiagram.ecore файла, который был получен на основе UML-модели, которая была рассмотрена в предыдущем пункте параграфа. Следующий скриншот позволяет понять суть этой мета-модели:

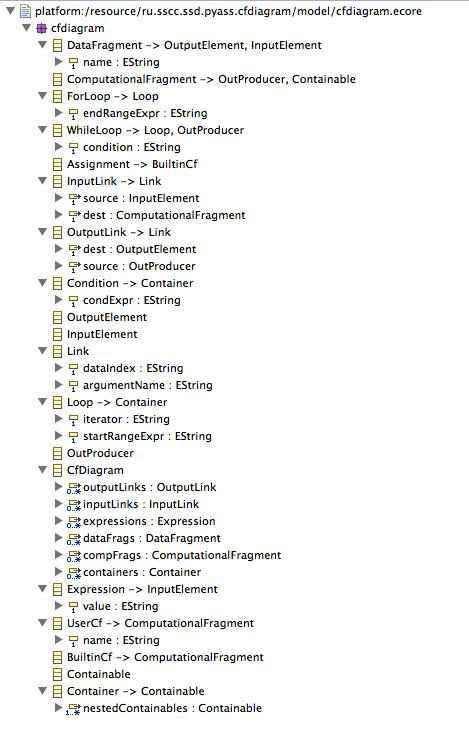


Рисунок 23 - представление мета-модели Visual LuNA

Все настройки, которые отображены в UML-виде, здесь задаются через специальное поле Properties:

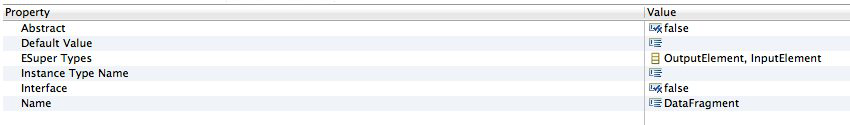


Рисунок 24 - свойства класса DataFragment

Помимо мета-модели Ecore, в Eclipse Modeling Framework есть другая мета-модель – GenModel (файл с расширением \*.genmodel). Мета-модель Ecore, которая была описана выше содержит информацию об определённых классах. Мета-модель Genmodel содержит дополнительную информацию для генерации кода, например информацию о файлах и путях к нем. Также эта метамодель содержит параметр, которым задается то как код должен быть сгенерирован. С точки зрения данной работы файл cfdiagram.genmodel не является интересным для рассмотрения, поэтому я приведу только скриншот для понимания:

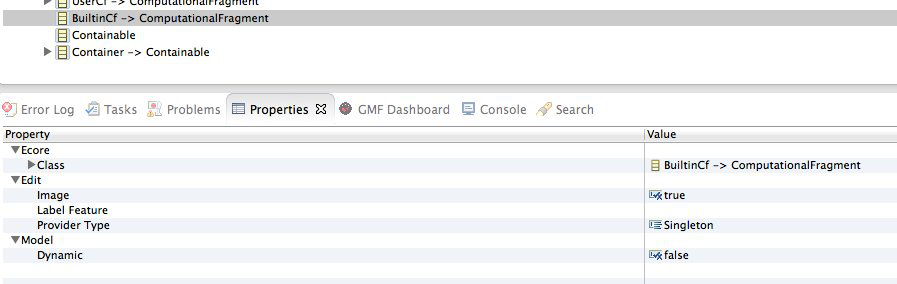


Рисунок 25 - представление класса BuiltinCF в мета-модели Genmodel

Вернемся к панели Graphical Modeling Editor, представленной на рисунке 22 – мы рассмотрели две основную модель графического редактора, теперь необходимо рассмотреть графическую модель, созданную при помощи Graphical Editing Framework.

Graphical Editing Framework позволяет задать графическое описание модели. Оно используется для определения фигур, связей и т.п. которые отображаются на диаграмме. Вначале задаются два файла - \*.gmfgraph (в данном случае cfdiagram.gmfgraph) и \*.gmftool (в данном случае cfdiagram.gmftool). Первый файл (.gmfgraph) используется для описания графических элементов модели (прямоугольник, эллипс, подпись поля (label), цвет и т.п.).

Так как файл получился достаточно большим приведу пример описания фигуры представляющей собой оператор for (рисунок 14):

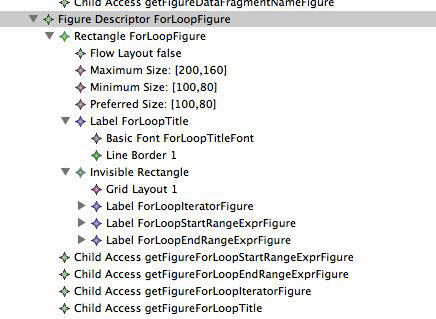


Рисунок 26 - описание фигуры «оператор for»

Рассмотрим подробнее:

1. Фигура представлена классом «прямоугольник»;
2. Тип слоя внутри фигуры – «плавающий»;
3. Задаются минимальный, максимальный, предпочитаемый размеры;
4. Добавляется подпись к этому прямоугольнику;
5. Задается шрифт (здесь уже стиль, размеры и т.п. задается через те же настройки «Properties»), текст и рисуется граница текста;
6. Внутри невидимый прямоугольник, в котором уже располагаются наш счётчик, начальное его значение и конечное;
7. Child Access – необходимы для получения доступа к описанным фигурам

Вернемся к нашей модели GMF, представленной на рисунке 22. Следующим шагом будет создание файла с расширением \*.gmftool (cfdiagram.gmftool). В этом файл задаётся описанная выше и представленная на рисунке 27 палитра инструментов. Здесь все просто, нужно просто добавить нужные инструменты, их названия и при желании иконки:

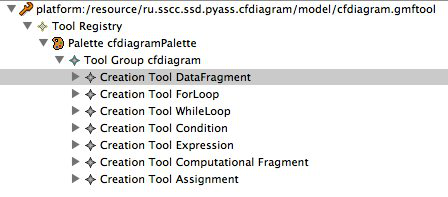


Рисунок 27 - представление палитры инструментов в GEF

После того как заданы графические элементы и палитра инструментов необходимо описать файл \*.gmfmap, представляющий собой Mapping Model (файл соответствия), который связывает модель, описанную при помощи Eclipse Modeling Framework, графическую модель, описанную в cfdiagram.gmfgraph, и модель палитры, описанную в cfdiagram.gmftool.

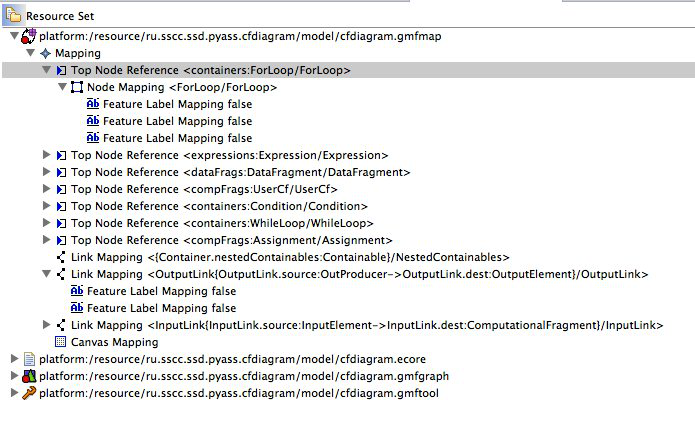


Рисунок 28 - файл соответствия в GEF

Top Node Reference – объекты (фрагмент данных, фрагмент вычислений, выражение, операторы for и while, условный оператор if). Link Mapping – ссылки (входящая и выходящая). Feature Label Mapping – подпись в объекте или ссылке.

После этого остаётся только нажать на «Transform» (см. рисунок 22) и на основе графического представления, палитры и связей между ними будет сгенерирован последний файл с расширением \*.gmfgen (в данном случае cfdiagram.gmfgen). Этот файл используется для генерации графического редактора. Таким образом на основе двух полученных файлов: cfdiagram.genmodel, используемого для генерации кода редактора) и cfdiagram.gmfgen, используемого для генерации кода графического редактора можно получить полностью рабочий редактор диаграмм (фрагментов кода).

### 3.2.4 Транслятор из Visual LuNA в LuNA

Вернёмся к целям исследования:

* Разработать визуальное представление фрагментированного алгоритма.
* Реализовать визуальную среду разработки фрагментированных программ.
* Реализовать транслятор для перевода графического представления алгоритма в представление на языке LuNA

Первые две уже реализованы. Остаётся последняя, заключительная часть – реализация транслятора с Visual LuNA в LuNA.

Транслятор был реализован в виде инкрементного компоновщика (incremental builder), для реализации которого платформа Eclipse предоставляет удобный интерфейс. При изменении файлов в проекте компоновщику поступает сигнал об изменении, содержащий информацию о том, какие файлы были изменены, и какие действия над ними были произведены (удаление, добавление, модификация). Используя эту информацию, компоновщик может осуществить сборку (трансляцию, компиляцию и т.п.) только тех файлов, которые подверглись изменению, не осуществляя лишний раз полную сборку проекта.

В нашей ситуации компоновщик транслирует, во-первых, EMF-модели, которые описывают диаграммы в .cfd файлах, и, во-вторых, объявления импортируемых фрагментов вычислений из файлов .imp, в байт-код языка LuNA. При возникновении ошибки на соответствующий файл устанавливается маркер с описанием ошибки и он не подвергается трансляции (сборка считается неуспешной). Байт-код LuNA представлен в формате JSON (JavaScript Object Notation). К сожалению, формальная спецификация формата этого байт-кода отсутствует, однако его структура была выведена путём эмпирического анализа (обратной инженерии). По окончанию сборки компоновщик обновляет файл *out.ja*, находящийся в корневой папке проекта и содержащий байт-код LuNA. Далее этот файл можно передать исполнительной системе LuNA для исполнения.

В реализации транслятора использовалась свободная библиотека google-gson[9], позволяющая (де)сериализовывать объекты Java в JSON-представление и конструировать JSON-объекты произвольного вида. Механизм конвертации Java-объектов использован не был, поскольку объекты в модели Visual LuNA существенно отличаются от их представления в LuNA.

Разберём пример простой программы на LuNA, ее представление в Visual LuNA и ее интерпретацию в gson:

import c\_show(string, int) as show;

sub main()

for i=1..5

show("It:", i\*100);

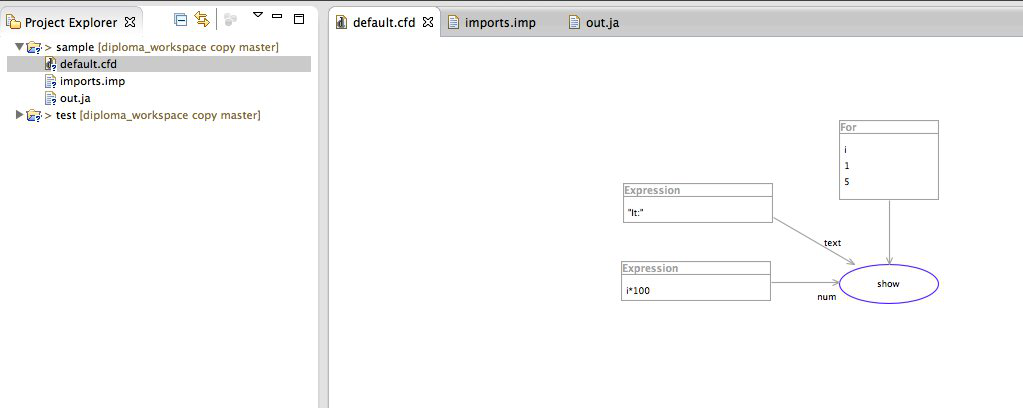


Рисунок 29 - представление программы в Visual LuNA

* default.cfd – файл с диаграммой (фрагмент кода)
* imports.imp – файл, содержащий объявления импортируемых атомарных фрагментов вычислений
* out.ja – байт-код LuNA

out.ja получился следующим:

{

"show" : {

   "type" : "extern",

   "code" : "c\_show",

   "args" : [{"type":"string"},{"type":"int"}]

},

"main" : {

   "type" : "struct",

   "args" : [],

   "body" : [

       {

           "type" : "for",

           "var" : "i",

           "first" : {"type":"iconst","value":1},

           "last" : {"type":"iconst","value":5},

           "body" : [

       {

           "type" : "exec",

           "id" : ["\_l4"],

           "code" : "show",

           "args" : [{"type":"sconst","value":"It:"},{"type":"\*","operands":[{"type":"id","ref":["i"]},{"type":"iconst","value":100}]}]

       }

           ]

       }

   ]

}

}

# ЗАКЛЮЧЕНИЕ

В результате работы был изучен подход фрагментированного программирования, проведено исследование систем визуального программирования и системы фрагментированного программирования LuNA. Было разработано визуальное представление фрагментированного алгоритма, на основе которого была реализована визуальная среда разработки фрагментированных программ Visual LuNA. После этого был реализован транслятор, который позволяет преобразовывать диаграммы, сделанные при помощи Visual LuNA в LuNA и запускать их.

На защиту выносится:

1. разработано визуальное представление фрагментированных алгоритмов;
2. реализован графический редактор фрагментированных программ Visual LuNA;
3. реализована возможность исполнения фрагментированных алгоритмов, описанных на Visual LuNA, при помощи JSON в байт-код понятный языку LuNA;
4. показана возможность исполнения фрагментированных алгоритмов, описанных в среде Visual LuNA.

**Дальнейшие планы**

[ToDo]

# Список использованных источников

1. Muhammed Al-Mulhem, Visual Programming Languages ICS 519 Part 1 // Sept. 15, 2002
2. Nan C. Shu, Visual Programming // Van Nostrand Reinhold, New York, NY. 1988
3. Margaret M. Burnett, Marla J. Baker, A Classification System for Visual Programming Languages // Journal of Visual Languages and Computing (1994) 5, 287-300
4. В.Л. Авербух, М.О. Бахтерев, Анализ средств визуального программирования параллельных вычислений // ИММ УрО РАН, г. Екатеринбург
5. VPL Introduction [Электронный ресурс]. – Режим доступа: <http://msdn.microsoft.com/en-us/library/bb483088.aspx>, свободный.
6. Microsoft Research. Kodu [Электронный ресурс]. – Режим доступа:

<http://research.microsoft.com/en-us/projects/kodu/>, свободный.

1. Eclipse Wiki. Graphical Modeling Framework [Электронный ресурс]. – Режим доступа: <http://wiki.eclipse.org/GMF>, свободный.
2. Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden, Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework // International Technical Support Organization, February 2004
3. Gson 2.2.4 API [Электронный ресурс]. – Режим доступа:

<http://google-gson.googlecode.com/svn/trunk/gson/docs/javadocs/index.html>, свободный.