Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Кафедра информатики
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ А.С. Кузнецов
подпись инициалы, фамилия
«13» 06 2017 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Инструментальная поддержка автоматической кодогенерации алгоритмически диверсифицированного программного обеспечения
тема

09.04.04 «Программная инженерия»
код и наименование направления

09.04.04.01 «Программное обеспечение вычислительной техники и автоматизированных систем»
код и наименование магистерской программы

Научный руководитель _____ 9.06.17 доцент, к.т.н А.С. Кузнецов
подпись, дата должность, ученая степень инициалы, фамилия

Выпускник _____ 9.06.17 М.В. Балушкин
подпись, дата инициалы, фамилия

Рецензент _____ 9.06.17 профессор, д.т.н С.В. Ченцов
подпись, дата должность, ученая степень инициалы, фамилия

Красноярск 2017

РЕФЕРАТ

Выпускная работа в форме магистерской диссертации по теме «Инструментальная поддержка автоматической кодогенерации алгоритмически диверсифицированного программного обеспечения» содержит 66 страниц, 50 рисунков, 26 приведённых использованных источников.

МУЛЬТИВЕРСИОННОЕ ПРОГРАММИРОВАНИЕ, NVP, КОДОГЕНЕРАЦИЯ, UML, ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ, АКТОРЫ

В работе рассматривается проблема автоматической кодогенерации мультиверсионного программного обеспечения. Проблема является актуальной в связи с тем, что данный момент не существует инструментария, позволяющего генерировать диверсифицированный программный код из графических моделей.

Объектом исследования является генерация программного кода из графической модели.

Цель диссертационного исследования состоит в разработке кодогенерационной модели для мультиверсионного программного обеспечения, а также реализации инструментария на её основе.

Поставленная цель достигается путём решения следующих задач:

- Разработка кодогенерационной модели мультиверсионного ПО,
- Разработка инструментария кодогенерации.

Основная идея работы заключается в разработке промежуточной модели представления программного обеспечения и применении модели акторов к мультиверсионному ПО.

Новизна работы заключается в разработке промежуточного представления программы, поддерживающего диверсификацию, а также применении модели акторов к мультиверсионному ПО.

В результате проведённой работы была разработана кодогенерационная модель мультиверсионного ПО и основанный на ней инструментарий.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ....5
1. Диверсификация....8
1.1. Виды диверсификации и сферы применения....8
1.2. Техники диверсификации....9
1.3. Вывод....12
2. Мультиверсионное программирование....13
2.1. Модель мультиверсионного программного обеспечения....13
2.2. Алгоритмы голосования....14
2.3. Примеры алгоритмов голосования....15
2.3.1. Усреднённое голосование....15
2.3.2. Формализованные алгоритмы голосования....15
2.2.3. Классические алгоритмы голосования....16
2.3. Вывод....18
3. Модель акторов и её применение к NVP....19
3.1. Модель акторов....19
3.2. Применение модели акторов к NVP....22
4. Кодогенерационная модель....23
4.1. Исходная модель: диаграммы деятельности....23
4.1.1. Язык описания диверсифицированных вызовов....28
4.2. Промежуточная модель....30
4.3. Процесс кодогенерации....34
4.4. Вывод....39
5. Разработка инструментария....39
5.1. Микрофреймворк Zmok....41
5.2. Классы утилит, модуль Hydra.Utils....43
5.3. Ядро системы, модуль Hydra.Core....44
5.3.1. Обобщённая модель графа....44
5.3.2. Модель диаграммы деятельности....45
5.3.3. Промежуточная модель....49
5.3.4. Модуль диверсификации: банк алгоритмов и контекст диверсификации....51
5.4. Редактор....52

5.4.1. Система событий.....52
5.4.2. Модификация редактора графов.....55
5.4.3. Пример работы программы.....60
5.5. Вывод.....62
ЗАКЛЮЧЕНИЕ.....63
СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ.....64

ВВЕДЕНИЕ

В современном информационном мире, когда активно автоматизируются многие сферы человеческой деятельности, надёжность становится критическим свойством программного обеспечения. Незапланированный отказ ПО может привести к потере доходов (например, из-за простоя сайта интернет-магазина или биржевого робота), а то и вовсе гибели большого числа людей (из-за отказа АСУ АЭС или сбои в ПО медицинского оборудования [1]).

Один из способов обеспечения надёжности — избыточность. То есть запуск нескольких копий программы, работающих параллельно. Таким образом, отказ одной копии не вовлечёт за собой отказ всей системы. Однако у такого подхода есть один недостаток — если в системе изначально была заложена ошибка, то высока вероятность, что все копии откажут одновременно. Этого можно избежать с помощью мультиверсионного программирования (N-Version Programming, NVP) [2].

Мультиверсионное программирование – способ обеспечения избыточности программного обеспечения, при котором все экземпляры ПО (либо его модулей) отличаются друг от друга, то есть диверсифицированы.

Традиционный подход к диверсификации ПО, при котором каждую версию разрабатывает отдельная команда, весьма дорогостоящ, а в случае, когда диверсификация применяется как способ повысить защищённость ПО, и вовсе бессмысленна. Автоматическая диверсификация ПО и модулей из спецификаций или моделей позволяет избавиться от этих трат.

Существующие системы диверсификации:

- Дополнения (skins) к Valgrind, например, RISE (Randomized Instruction Set Emulation),
- Некоторые DRM-системы, например, Denuvo,
- Project Diversify. Набор инструментов, разработанный INRIA и IRISA.

В работе рассматривается проблема генерации программного кода по модели с учётом специфики диверсифицированного программного обеспечения.

Проблема является актуальной в связи с тем, что практически не существует инструментов автоматизированной кодогенерации, учитывающих специфику мультиверсионного программирования, а именно диверсификацию и стратегию выбора (алгоритм голосования). Существует лишь небольшое количество таких работ: Project Diversify, разрабатываемый французскими институтами INRIA и IRISA, который не поддерживает NVP, и инструмент, разрабатываемый Д.В. Грузенкиным, который на данный момент находится в разработке [3].

Объектом диссертационного исследования является мультиверсионное программное обеспечение.

Предмет исследования – генерация программного кода с учётом диверсификации.

Цель диссертационного исследования состоит в разработке инструментария кодогенерации мультиверсионного программного обеспечения из графической модели описания (UML диаграмма деятельности).

Научная новизна работы заключается в разработке кодогенерационной модели, а также применении модели акторов к NVP.

В главе 1 описывается процесс диверсификации, её виды, сферы применения и техники диверсификации программного обеспечения.

В главе 2 рассматривается методология мультиверсионного программирования. Рассматриваются алгоритмы голосования такие, как усреднённое голосование, формализованные алгоритм голосования абсолютным большинством и алгоритм голосования абсолютным большинством.

В главе 3 описывается модель акторов, а также предлагается модифицированная структурная модель мультиверсионного программного обеспечения с применением акторов.

В главе 4 рассматривается процесс кодогенерации, ограниченное подмножество диаграммы деятельности UML, описывается разработанная промежуточная модель.

В главе 5 описывается разработанный инструментарий для генерации диверсифицированного программного кода из диаграммы деятельности.

1. Диверсификация

Диверсификация программного обеспечения — процесс создания функционально эквивалентных, но различных внутренне версий одной программной системы [4].

В основе диверсификации лежит модель швейцарского сыра: каждый слой – это версии, а дыры в нём – дефекты и уязвимости. Таким образом, дефект в одной версии компенсируется отсутствием такового в остальных (либо части из них) [5]. Модель швейцарского сыра представлена на рисунке 1.

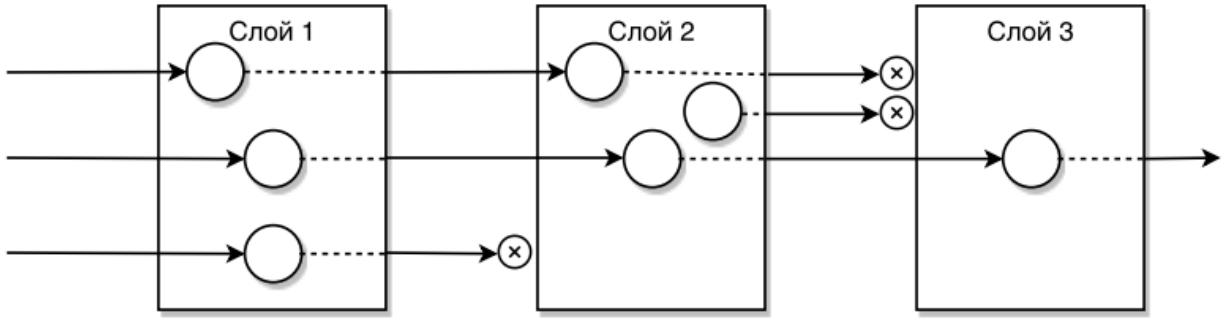

Рисунок 1 – модель швейцарского сыра

1.1. Виды диверсификации и сферы применения

Всего выделяют два вида диверсификации [6, 7]: автоматическая и управляемая.

При автоматической диверсификации различие между версиями программной системы достигается с помощью автоматических средств, тем или иным способом генерирующих новые версии программного обеспечения. Например, рандомизация запросов к базе данных, внедрение нефункционального кода и т.д.

При управляемой диверсификации различие между версиями программной системы достигается за счёт изменчивости окружения (операционная система, аппаратное обеспечение и т.д.) или целенаправленной разработки нескольких версий программной системы.

Основные области применения диверсификации: отказоустойчивость (fault-tolerance) [2] и защищённость (security) [6].

1.2. Техники диверсификации

Техники автоматической диверсификации в своём большинстве представляют собой всевозможные преобразования структуры программы и данных преимущественно случайным образом [8].

Наиболее важными преобразованиями являются [9]:

Преобразования на уровне инструкций: замена инструкции на эквивалентную последовательность (см. рисунок 1а), переупорядочивание инструкций (см. рисунок 1б), вставка нефункционального (мусорного) кода (см. рисунок 2).

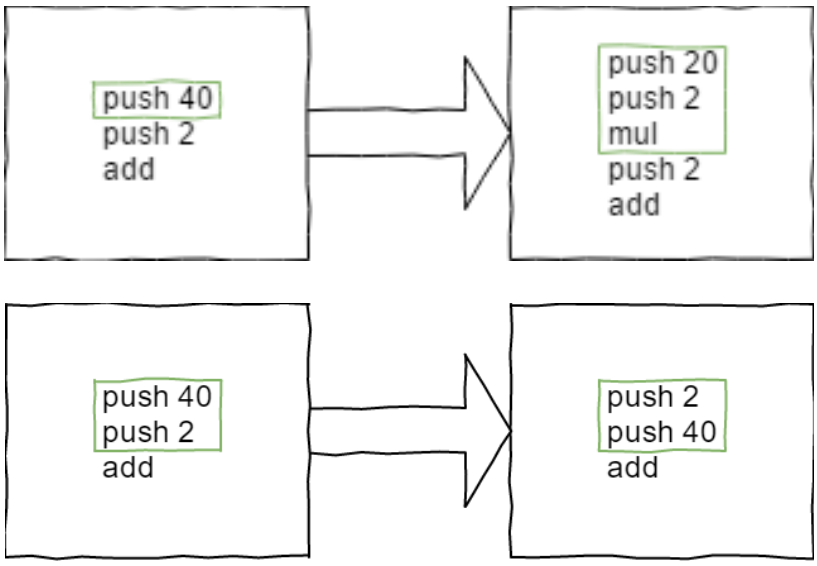

Рисунок 2 – а) эквивалентная последовательность, б) переупорядочивание инструкций

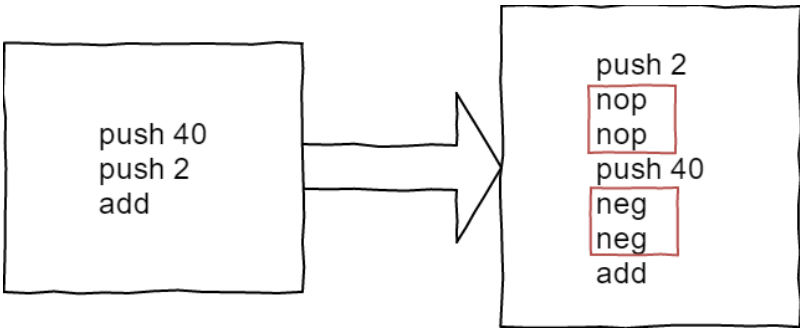

Рисунок 3 – а) эквивалентная последовательность, б) переупорядочивание, в) мусорные инструкции

Преобразования на уровне функций: рандомизация расположения данных на стеке (отступ, направление стека), переупорядочивание параметров вызова (см. рисунок 3), инлайнинг и разбиение функции на несколько частей (см. рисунок 4).

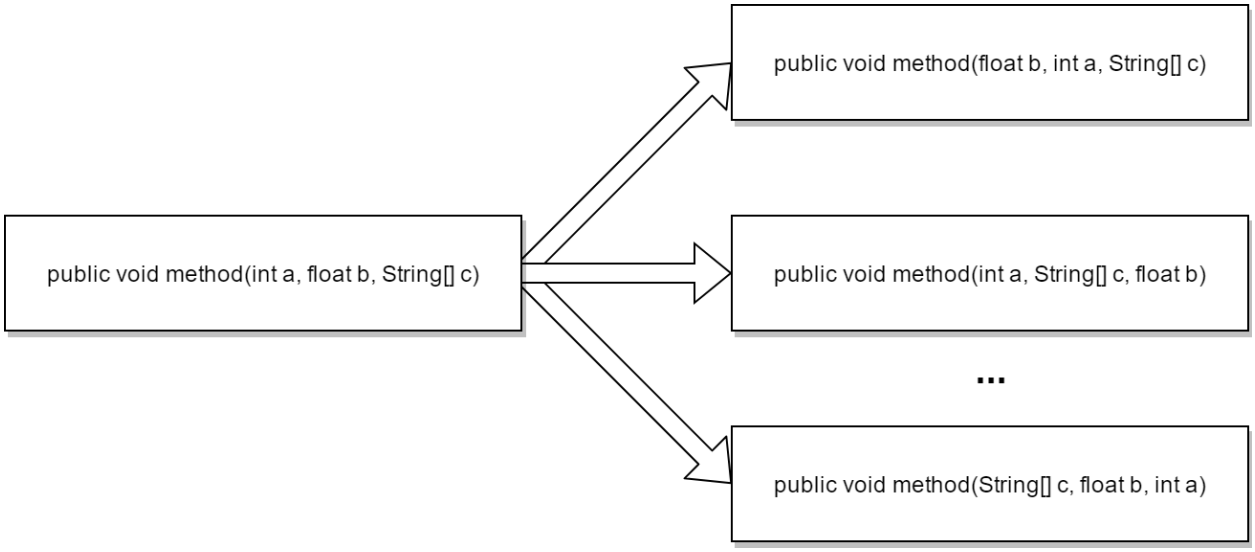

Рисунок 3 – переупорядочивание аргументов функции

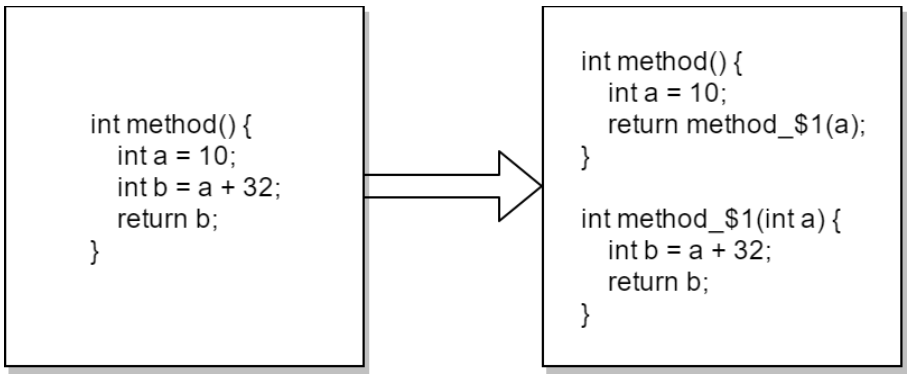

Рисунок 4 – разбиение функции

Преобразования на программном уровне: переупорядочивание функций внутри исполняемого файла либо подключаемой библиотеки, рандомизация размещения адресного пространства (ASLR).

Преобразования на уровне данных: рандомизация расположения статичных данных, шифрование на основе случайно сгенерированного ключа, рандомизация структур данных (расположения полей) и кучи (добавление случайного сдвига, разделение кучи на несколько регионов и т.д.).

Техники управляемой диверсификации можно разделить на следующие группы: управляемые естественные, управляемые функциональные и структурные.

Управляемая естественная диверсификация включает в себя следующие техники: конфигурирование программной системы (поведение системы может меняться в соответствии с заданной конфигурацией), естественное разнообразие окружения (операционная система, брандмауэр, виртуальные машины, сервера приложений, аппаратное обеспечение и т.д.).

К управляемым функциональным техникам диверсификации относятся: диверсификация на основе классов, диверсификация с помощью плагинов.

Структурная диверсификация включает в себя: блоки восстановления и мультиверсионное программирование.

Блок восстановления состоит из набора версий программного модуля (альтернатив), выполняющихся последовательно и снабжённых блоком принятия решения о корректности работы версии. В случае, если одна из версий завершается с ошибкой либо её результат не проходит приёмочный тест (блок принятия решения), состояние системы сбрасывается до начального и запускаются следующая альтернатива в списке [10]. Данный процесс изображён на рисунке 5.

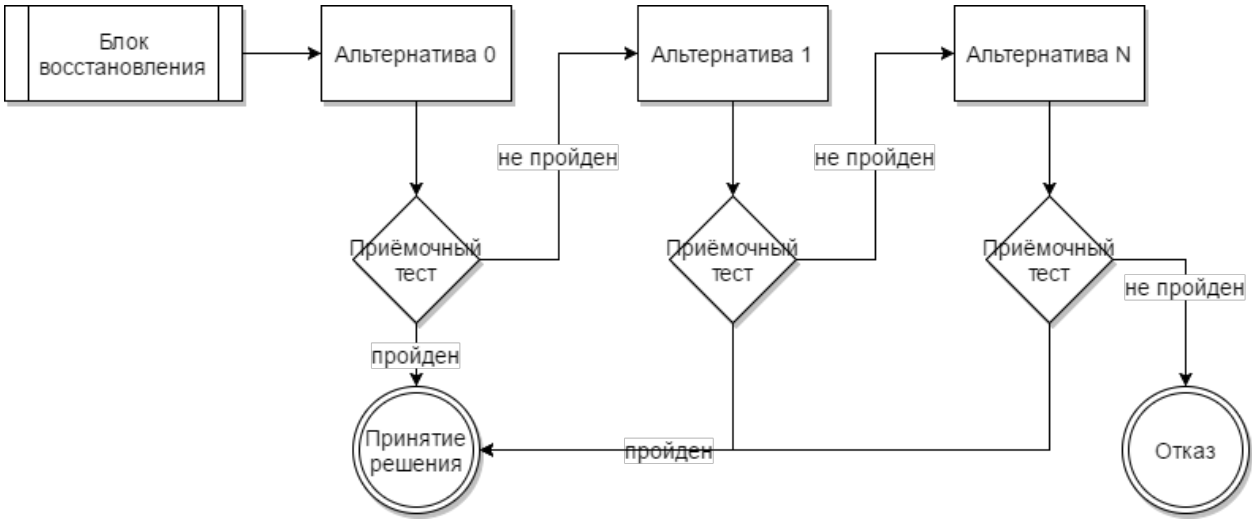

Рисунок 5 – блок восстановления

Мультиверсионное программирование описано в главе 2 «Мультиверсионное программирование».

1.3. Вывод

Большинство описанных выше приёмов и техник применимы только для диверсификации в целях обеспечения защищённости программной системы. Например, большая часть техник автоматической диверсификации основана на случайных неконтролируемых преобразованиях, что мало пригодно для обеспечения отказоустойчивости системы.

Управляемая же диверсификация пригодна для обоих случаев. Например, в разных операционных системах – разные уязвимости и дефекты.

2. Мультиверсионное программирование

Мультиверсионное программирование (N-version programming) — способ обеспечения избыточности программного обеспечения, при котором все экземпляры ПО (либо его модулей) отличаются друг от друга, то есть диверсифицированы [11, 2].

2.1. Модель мультиверсионного программного обеспечения

Главными компонентами мультиверсионного программного обеспечения являются: две или более версии программного модуля, а также блок принятия решения (алгоритм голосования) о результатах работы программы. Структурная модель мультиверсионного ПО представлена на рисунке 6 [12].

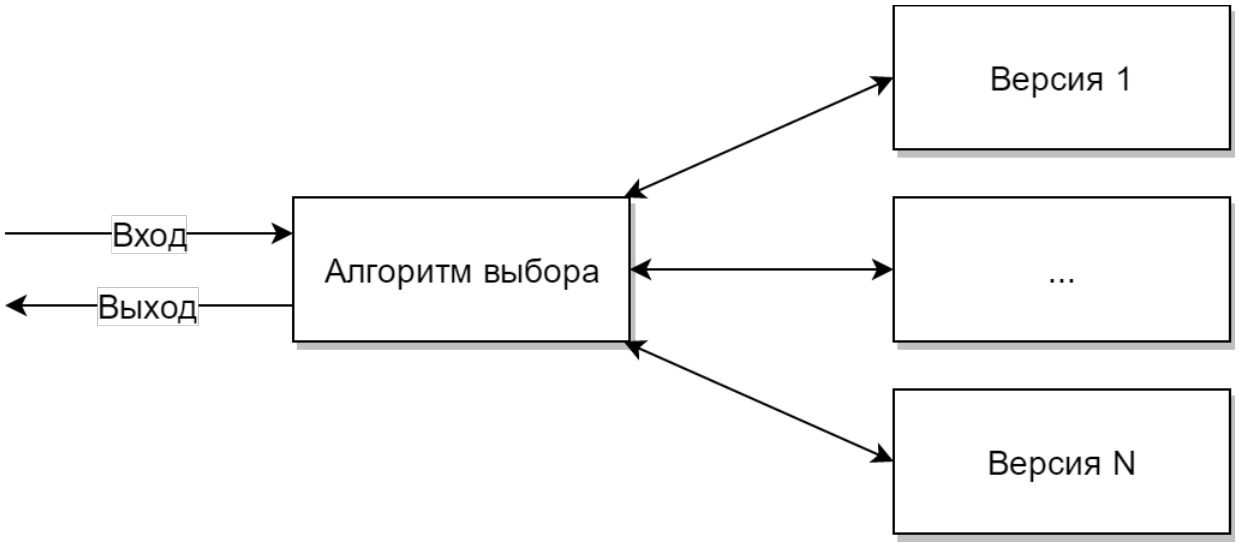

Рисунок 6 – структурная модель мультиверсионного ПО

Процесс работы мультиверсионного ПО происходит следующим образом:

1) Последовательное либо параллельное (в зависимости от специфики реализации) исполнение версий диверсифицированного модуля,

2) На основе результатов версий с помощью одного из алгоритмов голосования (см. 2.2) выносится решение о результате модуля в целом.

2.2. Алгоритмы голосования

Все алгоритмы голосования можно разбить на несколько групп [11]:

1) алгоритмы, принимающие решения вне зависимости от схожести выходных данных: алгоритм максимального правдоподобия (MLV), усреднённое голосование,

2) алгоритмы, принимающие решения на основе сравнения выходных данных:

2.1) формализованные: формализованный алгоритм голосования абсолютным большинством (FMV), формализованный алгоритм голосования согласованным большинством (FCV),

2.2) неформализованные:

2.2.1) классические: алгоритм голосования абсолютным большинством (NVP-MV), алгоритм голосования согласованным большинством (NVP-CV),

2.2.2) нечёткие: нечёткий алгоритм голосования абсолютным большинством (Fuzzy MV), нечёткий алгоритм голосования согласованным большинством (Fuzzy CV),

2.2.3) с минимизацией: алгоритм голосования согласованным большинством с минимизацией (Min CV), алгоритм голосования абсолютным большинством с минимизацией (Min MV).

2.3. Примеры алгоритмов голосования

2.3.1. Усреднённое голосование

Алгоритм усреднённого голосования относится к первой категории алгоритмов голосования – он не зависит от схожести выходных данных версий.

Решение определяется следующим образом:

$$X = \frac{1}{N} \sum_{i=1}^{N} x_i,$$

где X – решение, N – число версий, а x_i – выход *i*-й версии.

2.3.2. Формализованные алгоритмы голосования

Особенностью формализованных алгоритмов голосования является то, что в процессе поиска решения множество выходов разбивается на подмножества, при этом некоторые выходу могут входить более чем в одном такое подмножество.

Формализованный алгоритм голосования абсолютным большинством (FMV):

N – число версий, x_i – выход *i*-й версии, а ε – допустимое отклонение.

На 1-м шаге алгоритма происходит разбиение множества версий на N подмножеств, где каждому выходу соответствует своё подмножество. Для выхода *i* подмножество строится следующим образом:

$$C_i = \left(x_j \mid \forall j,\ \left(x_i - x_j \right) \le \varepsilon \right),\ \ j = 1 \ldots N$$

На 2-м шаге определяется набор корректных выходов. Пусть $len(C_i)$ – количество элементов в подмножестве C_i, тогда, если существует такое подмножество, для которого справедливо неравенство

$$len(C_i) \ge ceil\left(\frac{N+1}{2} \right),$$

то это подмножество является решением. Если такое подмножество отсутствует, то принять решение с помощью данного алгоритма невозможно.

Формализованный алгоритм голосования согласованным большинством (консенсус):

N – число версий программного модуля, x_i – значение выхода i-й версии, а ε – допустимое отклонение.

На 1-м шаге алгоритма происходит разбиение множества версий на N подмножеств, где каждому выходу соответствует своё подмножество. Для выхода i подмножество строится следующим образом:

$$C_i=\left(x_j \mid \forall j,\ \left(x_i-x_j\right)\le\varepsilon\right),\ \ j=1\ldots n$$

На 2-м шаге определяется набор корректных входов. Пусть $len\left(C_i\right)$ – количество элементов в подмножестве C_i, тогда выбирается такое подмножество, для которого $len\left(C_i\right)=max$. Если таких подмножество несколько, то результат выбирается случайным образом из всех таких подмножеств.

2.2.3. Классические алгоритмы голосования

Неформализованные алгоритмы голосования основаны на разбиении множества выходов версий на непересекающиеся подмножества.

Алгоритм голосования абсолютным большинством работает следующим образом:

N – количество версий модуля, x_i – выход i-й версии, а ε – допустимое отклонение.

Шаг 1. Построение матрицы согласования R размерности $N\times N$, элементы которой вычисляются по следующему принципу:

$$r_{ij}=\begin{pmatrix}1,\ если\ (x_i-x_j)\le\varepsilon\\0,\ если\ (x_i-x_j)>\varepsilon\end{pmatrix}$$

Шаг 2. Проверка свойства транзитивности матрицы согласования:

$$\frac{r_{ik}=1,\ r_{jk}=1}{r_{ik}=1},\ \forall i,\ j.$$

Если свойство транзитивности не соблюдается, то матрицу согласования необходимо изменить с помощью булевых композиций до тех пор, пока отношение не будет удовлетворено.

Булева композиция:

Пусть даны матрицы согласования *A* и *B*, а их элементы $a_{ij}\in\{0,\ 1\}$ и $b_{ij}=(0,\ 1)$, тогда булева композиция этих матриц представляет собой:

$$C=A\circ B,\quad c_{ij}=\oplus_{k=1}^{N}(a_{ik}\otimes b_{kj}),$$

где $\oplus$ – функция логического «ИЛИ», а $\otimes$ – функция логического «И».

Для достижения условия транзитивности необходимо выполнить последовательное выполнение булевых композиций матрицы согласования самой с собой по принципу:

$$E=R_1\cup R_2\cup\ldots\cup R_{N-1},\ где\ R_2=R\circ R,\ R_3=R\circ R\circ R\ \ И\ \ т.д.$$

Шаг 3. Выбор решения. Пусть Y_i – количество единиц в *i*-й строке матрицы согласования, тогда в качестве решения выбирается строка, удовлетворяющая условию:

$$Y_i\ge ceil\left(\frac{N+1}{2}\right).$$

Алгоритм голосования согласованным большинством:

Шаг 1. Строится матрица согласования R.

Шаг 2. Проверяется отношение эквивалентности матрицы R, при необходимости вычисляется булева композиция.

Шаг 3. Выбор решения. Пусть Y_i – количество единиц в *i*-й строке матрицы согласования, тогда в качестве решения выбирается строка, в которой Y_i максимально.

2.3. Вывод

Мультиверсионное программное обеспечение позволяет достичь высоких показателей надёжности программного обеспечения. Однако такой подход к разработке не лишён недостатков, а именно:

- Дороговизна разработки,
- Высокие требования к процессу разработки и проектирования/

Все эти недостатки достаточно очевидны – разработка мультиверсионного ПО требует в несколько раз больше денег, чем разработка обыкновенного ПО, а неточности, допущенные при разработке требований, могут привести к ещё большим затратам.

Отдельно стоит отметить схожесть модели мультиверсионного ПО и модели масштабируемой распределённой системы [13]. Фактически, мультиверсионное ПО – это частный случай распределённой системы, где все инстансы отличаются друг от друга внутренне (диверсифицированы), а в качестве монитора выступает алгоритм голосования. Например, С. Алльe комбинирует данные модели в работе «Multi-tier diversification in Web-based software application» [6], используя диверсифицированные инстансы Web-приложения и балансировщик нагрузки в качестве монитора.

3. Модель акторов и её применение к NVP

В 1973 году К. Хьюит, П. Бишоп и Р. Штайгер предложили новый поход к реализации конкурентных и параллельных вычислений на основе независимых, взаимодействующих друг с другом посредством сообщений вычислительных агентов, называемых акторами [14].

3.1. Модель акторов

Акторы – вычислительные агенты, существующие независимо друг от друга. Под независимостью подразумевается отсутствие общих ресурсов. Так как акторы не имеют разделяемой памяти, в системах, основанных на данной модели, отсутствуют проблемы свойственные многопоточным системам. Например, состояние гонки или взаимные блокировки.

Взаимодействие между акторами происходит посредством сообщений. В ответ на входящее сообщение актор может [14, 15]:

- отправить конечное число сообщений другим акторам,
- изменить своё поведение для обработки следующего сообщения,
- создать конечное число новых акторов,
- завершить свою работу.

Важной частью системы акторов является так называемый «почтовый ящик» [16] – асинхронная очередь сообщений, ассоциированная с одним актором и имеющая уникальный адрес.

Сообщения, с помощью которых коммуницируют акторы, также называются задачами. Каждая задача состоит из трёх частей:

- уникальный идентификатор, позволяющий различать задачи,
- «почтовый адрес» получателя,
- непосредственно сами данные.

Пример работы системы из двух акторов, представляющих собой некоторое хранилище данных.

Сообщения, принимаемые актором *A*:

- save(object) – записывает переданный объект в память,
- commit – передаёт все записанные объекты актору *B* и удаляет из памяти актора *A*.

Сообщения, принимаемые актором *B*:

- save(object[]) – записывает переданные данные в память,
- read – возвращает все данные актора *B* и очищает свою память.

Шаг 1. Актору A посылается сообщение save(42):

$$A = (\ 42\ ),\ B = \varnothing,$$

где запись вида $X = (i)$ представляет содержимое актора *X*.

Шаг 2. Актору A посылается сообщение save(451):

$$A = (42,\ 451),\ B = \varnothing,$$

Шаг 3. Актору A посылается сообщение commit, актор A посылает сообщение актору B save([42, 451]):

$$A = \varnothing,\ B = (42,\ 451),$$

Шаг 4. Актору B посылается сообщение read:

$$A = \varnothing,\ B = \varnothing.$$

Описанное взаимодействие представлено на рисунке 7.

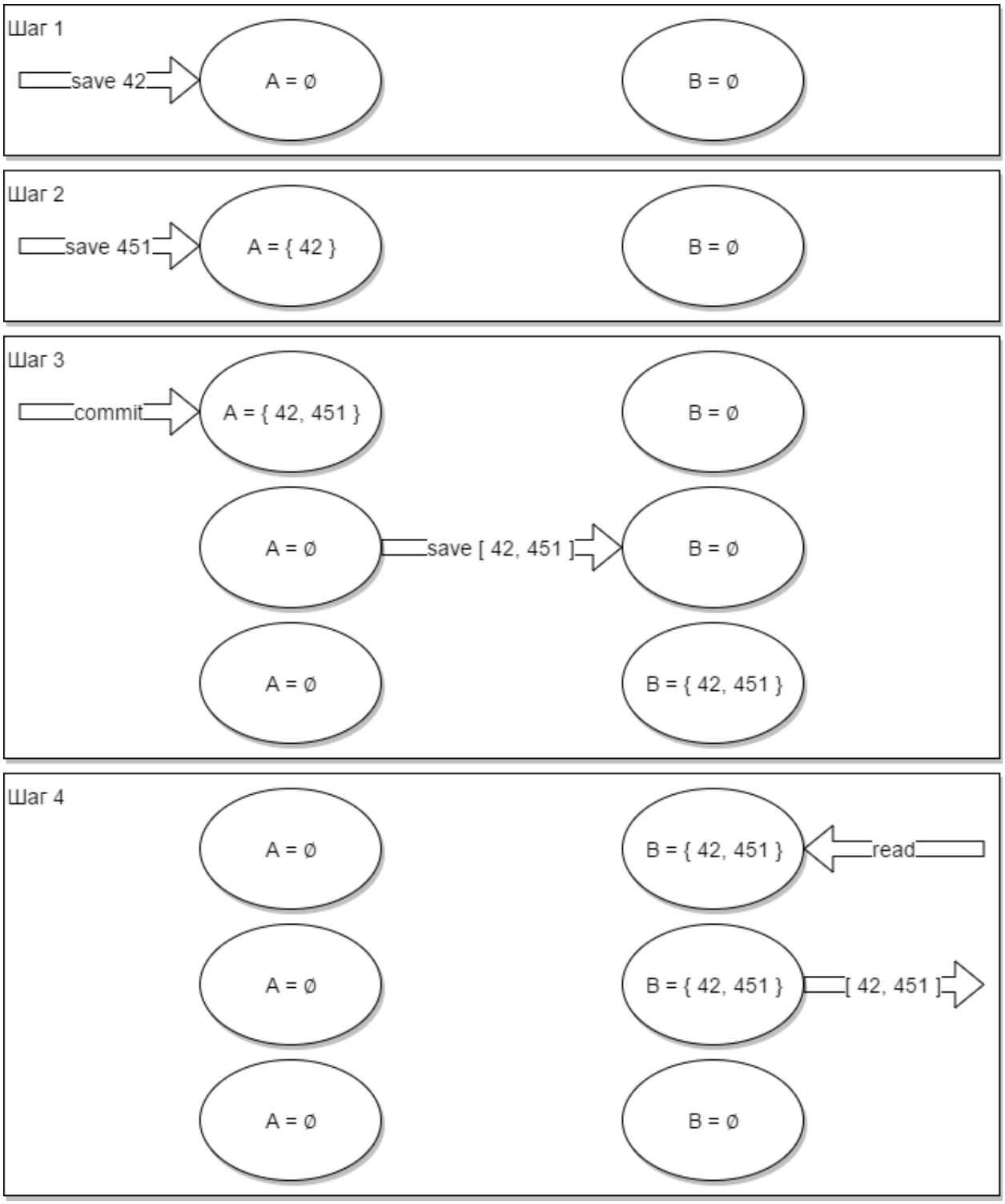

Рисунок 7 – пример взаимодействия акторов

Особенности модели акторов (независимость акторов, коммуникация через сообщения) позволили ей найти широкое применение в разработке распределённых масштабируемых систем. Например, язык программирования Erlang применяется для реализации

телекомуникационных протоколов, а библиотека Akka – в анализе больших данных (Big Data).

3.2. Применение модели акторов к NVP

Как было описано выше (см. 2.3), модель мультиверсионного программного обеспечения представляет частный случай модели распределённого ПО, поэтому её можно с легкостью реализовать в виде системы акторов следующим образом:

- Каждый компонент мультиверсионного модуля реализуется в виде отдельного актор,
- Алгоритм голосования также представляется в виде актора.

Модифицированная структура мультиверсионного ПО представлено на рисунке 8.

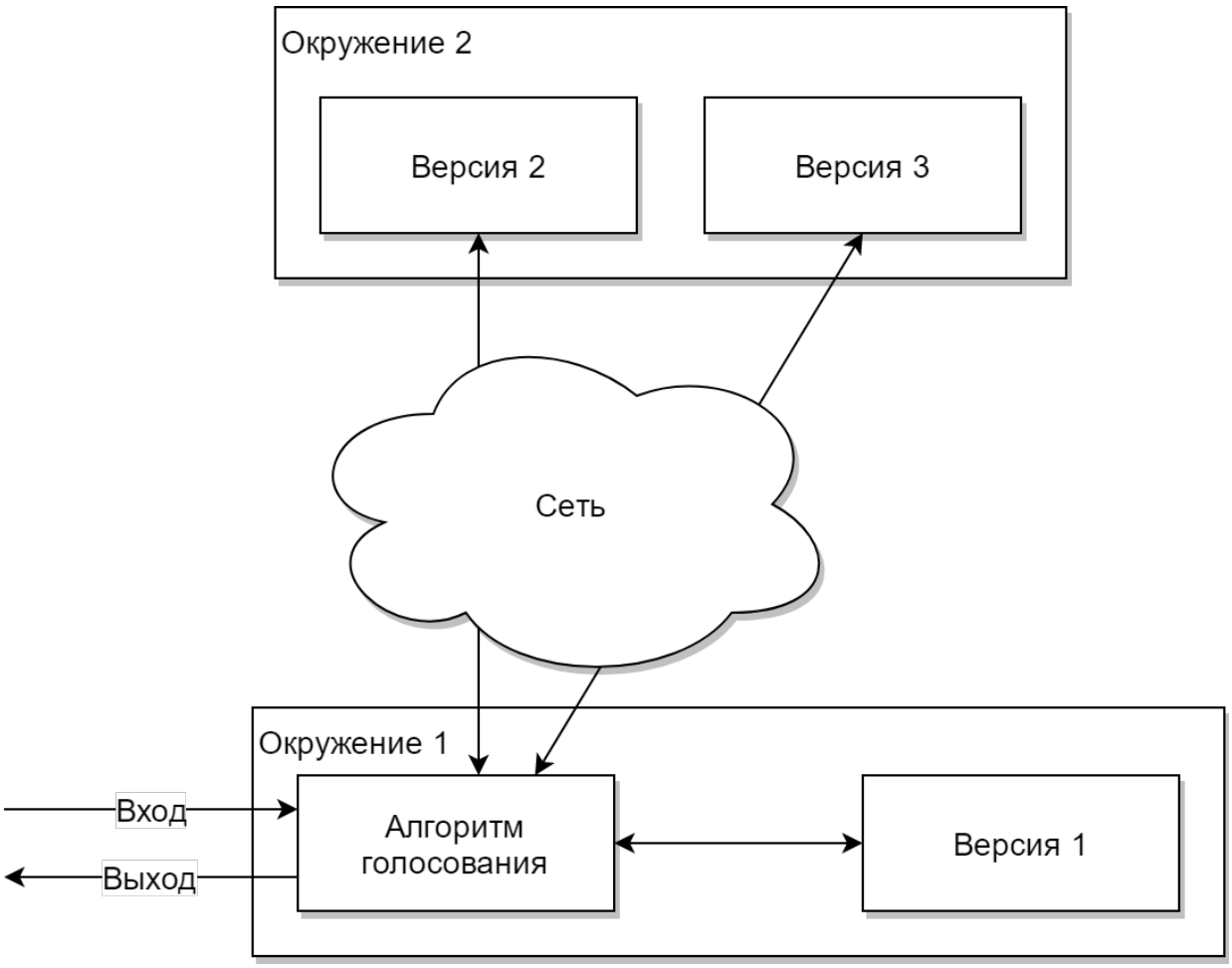

Рисунок 8 – модифицированная структурная модель муьтиверсионного ПО

Полученная архитектура мультиверсионного программного обеспечения обладает следующими особенностями:

- Позволяет значительно ускорить процесс разработки ПО за счёт использования уже существующего инструментария (например, библиотеки Akka, Pulsar/Quasar),
- Позволяет разрабатывать распределённые системы, что в свою очередь даёт возможность применить диверсификацию на уровне аппаратного обеспечения.

4. Кодогенерационная модель

В данной главе описывается разработанная модель генерации алгоритмически диверсифицированного исходного кода на Java.

В разделе 4.1 рассматривается исходная модель – UML диаграммы деятельности,

В разделе 4.2 описана разработанная промежуточная модель,

В разделе 4.3 описан процесс кодогенерации.

4.1. Исходная модель: диаграммы деятельности

Диаграммы деятельности – диаграмма UML, которая применяется для моделирования и проектирования динамических аспектов, то есть поведения, программных систем [17, 18].

Диаграмма деятельности показывает поток управления между деятельностями.

Деятельность (activity) – набор последовательно либо параллельно выполняющихся действий (action), каждое из которых может изменить состояние системы. Действие заключается в выполнении вычислений, создании либо уничтожении объекта, вызове другой операции (деятельности), посылке сигнала и т.д. Примеры действий представлены на рисунке 9.

Узел деятельности (activity node) – организационная единица деятельности, представляющая собой вложенную группу действий или других узлов. Узлы деятельности можно рассматривать в контексте проектирования программной системы как функцию либо метод. Примеры узлов деятельности представлены на рисунке 10.

Рисунок 9 – действия

Рисунок 10 – узлы деятельности

При завершении выполнения действия или узла деятельности, поток управления немедленно переходит к следующему узлу. Поток изображается в виде направленной стрелки от одного узла к другому. Чтобы обозначить начало и конец потока управления используются специальные узлы: инициализация и завершение. Пример потока управления приведён на рисунке 11.

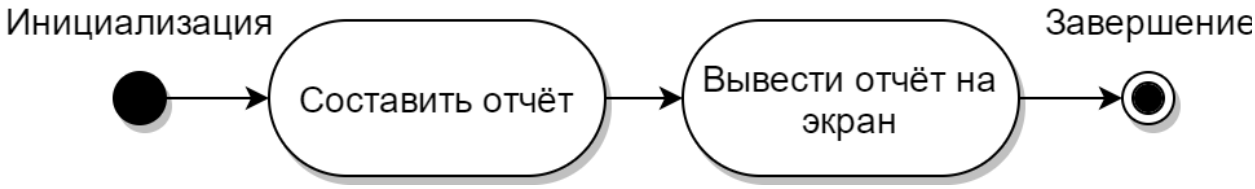

Рисунок 11 – поток управления

Для спецификации альтернативных путей выполнения, выбираемых с помощью логических выражений, используются узлы ветвления (decision) и объединения (merge).

Узел ветвления может содержать один входящий поток и несколько исходящих, на которых помещается логическое выражение (guard), вычисляемое при входе в ветвь. Условия исходящих потоков должны учитывать все возможные варианты, а также не должны пересекаться. Для объединения двух путей потока используется узел объединения. Пример ветвления представлен на рисунке 12.

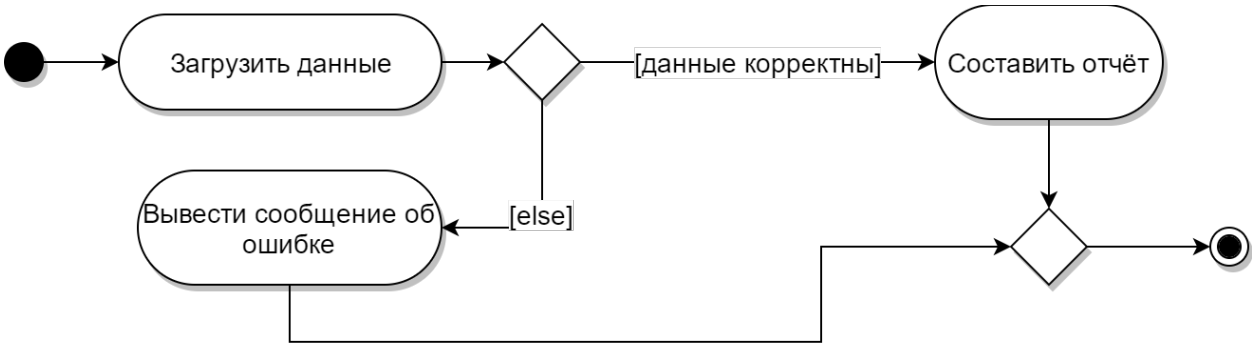

Рисунок 12 – ветвление

Основное отличие диаграмм деятельностей блок-схем – возможность изобразить параллельные потоки управления. Для этого используются линейка синхронизации (synchronization bar).

Разделение представляет собой расщепление (fork) одного потока управления на несколько параллельных либо конкурентных. Для слияния двух потоков управления используется соединение (join). Пример параллельного потока представлен на рисунке 13.

Рисунок 13 – параллельные потоки управления

Для организации узлов в диаграмме деятельности используются «плавательные дорожки» (swimlanes). Каждая дорожка имеет уникальное имя. Узлы могут принадлежать только одной дорожке одновременно. Пример использования «плавательных дорожек» представлен на рисунке 14.

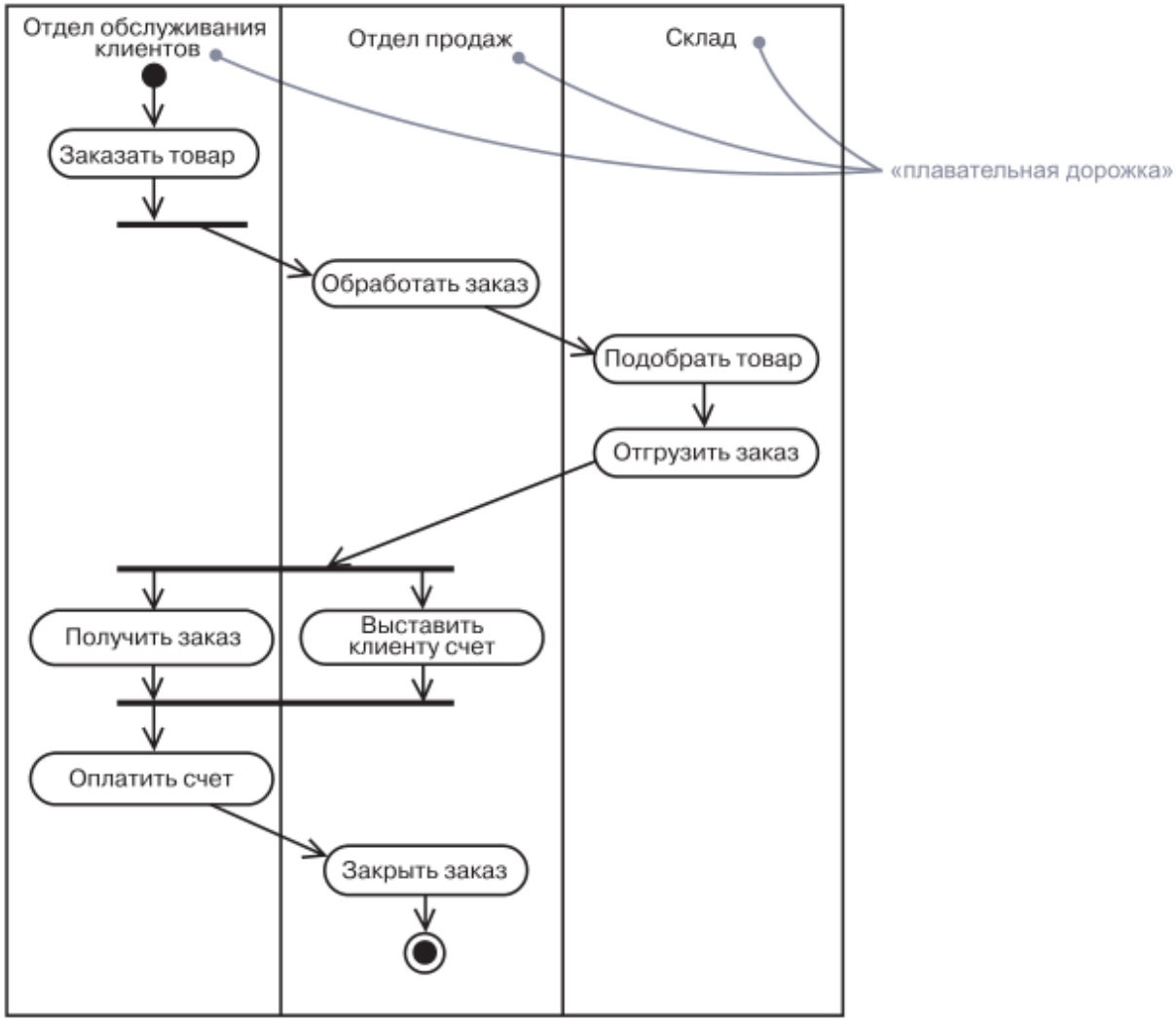

Рисунок 14 – «плавательные дорожки»

Помимо описанных выше элементов, диаграммы деятельности могут содержать следующие элементы [19]:

- объекты,
- отправка и получение сигналов,
- разбиения,
- области расширения.

В UML имеются три механизма расширения диаграмм [20]:

- *Стереотипы* (stereotypes) позволяют создавать новые элементы путём наследования от существующих для решения проблем специфичных в моделируемой области. Например, соединения между узлами могут помечаться для обозначения типа связи прототипами include и extends, а классы-исключения – прототипом Exception,
- *Помеченные значения* (tagged values) позволяют расширять свойства стереотипов путём включения новой информации в его спецификацию. Например, метки «версия» и «автор» для класса,
- *Ограничения* (constraints) расширяют семантику базового элемента, позволяя добавлять новые правила или модифицировать существующие. Например, поле «Баланс» класса «Счёт» может иметь ограничение «больше нуля». Также для записи ограничений используется специализированный язык ограничений Object Constraint Language [20].

В качестве исходной модели для кодогенерации было выбрано ограниченное подмножество диаграмм деятельности.

На модель накладываются следующие ограничения:

- Действия представляют собой исключительно исходный код на целевом языке программирования,
- Диверсифицируемое действие должно быть помечено стереотипом Diversified,
- Диверсифицируемые действия записываются в специальной форме (далее – язык описания диверсифицированных вызовов), описанной ниже.

Стереотип Diversified имеет следующую спецификацию (помеченные значения):

- используемый алгоритм голосования,
- используемый диверсифицированный алгоритм,
- используемые версии алгоритма.

4.1.1. Язык описания диверсифицированных вызовов

Так как на этапе кодогенерации нет доступа к информации о типах, выражение в диверсифицируемом вызове следует записывать на специальном языке выражений.

Для описания синтаксиса языка диверсифицированных вызовов воспользуемся контекстно-свободной грамматикой. Контекстно-свободная грамматика состоит из следующих компонентов [21]:

- Множество терминальных символов (токенов), представляющих элементарные единицы языка. Например, числа, строковые значения, идентификаторы и т.д.,
- Множество нетерминальных символов (синтаксических переменных), представляющих множество терминальных строк, заданное с помощью продукции,
- Множество продукций. Продукции состоят из нетерминалов в левой части и последовательности терминалов и/или нетерминалов в её правой части,
- Стартовый (начальный) нетерминальный символ.

Грамматика языка описания диверсифицированных вызовов имеет следующий вид:

- Символ: *Symbol ::= JavaIdentifierStart JavaIdentifier* (‘[]’)*, где *JavaIdentifierStart* – любой корректный символ, с которого может начинаться

Java-идентификатор [22], а *JavaIdentifier* – любой корректный символ Java-идентификатора,

- Оператор присваивания: *Assignment ::= Argument ‘=’ Funcall*,
- Аргумент (пара «имя, тип»): *Argument ::= Symbol ‘::’ Symbol*,
- Вызов функции: *Symbol ‘(‘ (Argument ‘,’)* Argument ‘)’*.

Графическое представление языка описания диверсифицированного вызова изображено на рисунках 15-18.

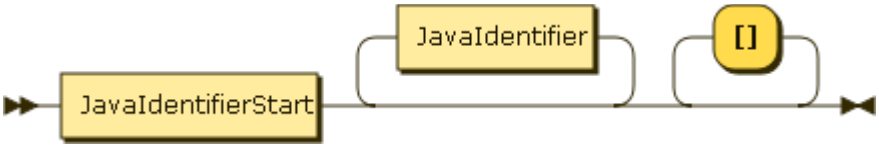

Рисунок 15 – синтаксическая диаграмма нетерминала Symbol

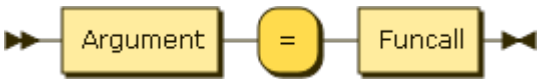

Рисунок 16 – синтаксическая диаграмма нетерминала Assignment

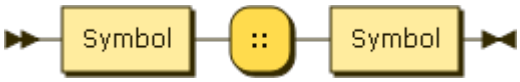

Рисунок 17 – синтаксическая диаграмма нетерминала Argument

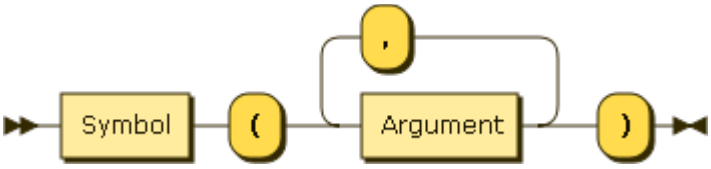

Рисунок 18 – синтаксическая диаграмма нетерминала Funcall

Примеры выражения на языке описания диверсифицированных вызовов:

- average :: double = average(array :: int[]),
- minByAge :: Person = findMinBetween(data :: Person[], min :: int, max :: int),
- min :: Double = optimize(goal :: Goal),
- sorted :: Integer[] = sort(data :: Integer[]).

4.2. Промежуточная модель

Промежуточная модель представляет собой дополнительный слой абстракции между исходной моделью и целевым языком программирования.

Главными преимуществами явно выделенного промежуточного представления являются:

- Простота генерации исходного кода. В отличие от диаграмм деятельности или блок-схем такое промежуточное представление изначально разрабатывается с целью последующей кодогенерации,
- Слой абстракции между исходной моделью и программным кодом. Использование промежуточной модели в процессе кодогенерации позволяет добавлять новые виды исходных моделей без изменения самого процесса генерации исходного кода программного обеспечения.

Промежуточная модель должна обладать следующими характеристиками:

- Не допускается наличие циклов,
- Контроль над потоком управления,
- Близость к абстрактному синтаксическому дереву.

Промежуточная модель представляет собой ацикличный ориентированный граф (дерево).

Действие (action), как и в диаграмме деятельности, заключается в выполнении вычислений, создании либо уничтожении объекта, вызове других операций, функций, посылке сигнала и т.д. Действие описывается в виде конкретного выражения либо последовательности выражений на целевом языке программирования. Примеры действий представлены на рисунке №19.

Диверсифицированные действия (diversified actions) представляют вызов мультиверсионного программного модуля. Примеры таких действий представлены на рисунке 20.

Контроль над потоком управления обеспечивается узлом последовательности действий, который представляет собой, как следует из названия, набор последовательно выполняющихся действий. Пример приведён на рисунке 21.

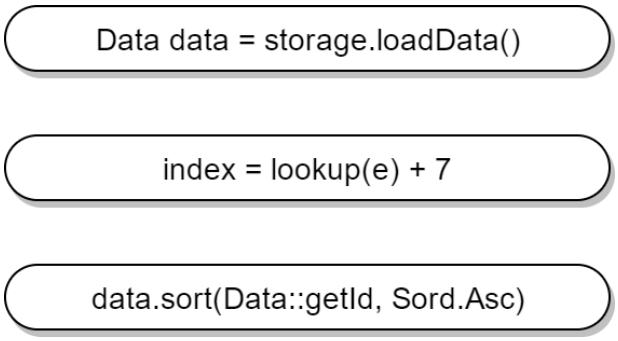

Рисунок 19 – действия

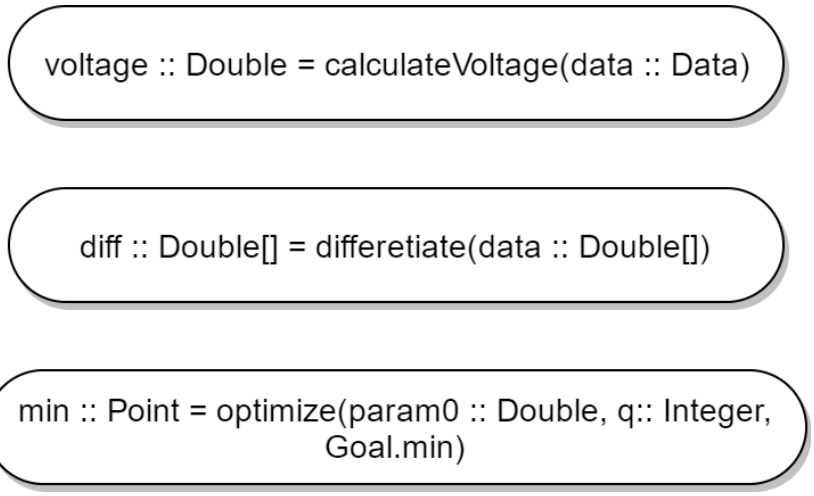

Рисунок 20 – диверсифицированные вызовы

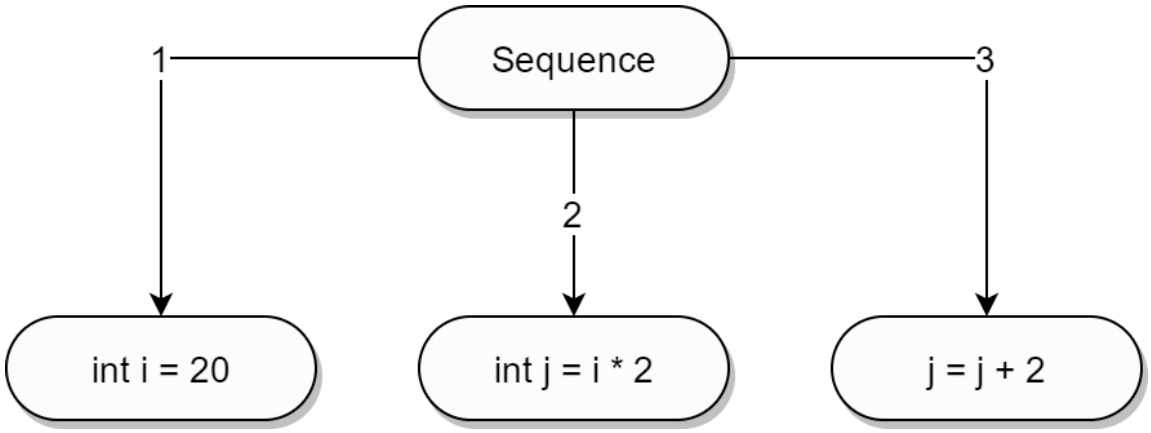

Рисунок 21 – последовательность действий

Для реализации ветвления используется условный оператор Conditional, представляющий собой узел, имеющий три именованных выхода (см. рисунок 22):

- Test – действие, представляющее условие ветвления,
- If-Branch – действие либо последовательность действий, представляющих True-ветвь условного оператор,
- False-Branch – действие либо последовательность действий, представляющих False-ветвь условного оператора.

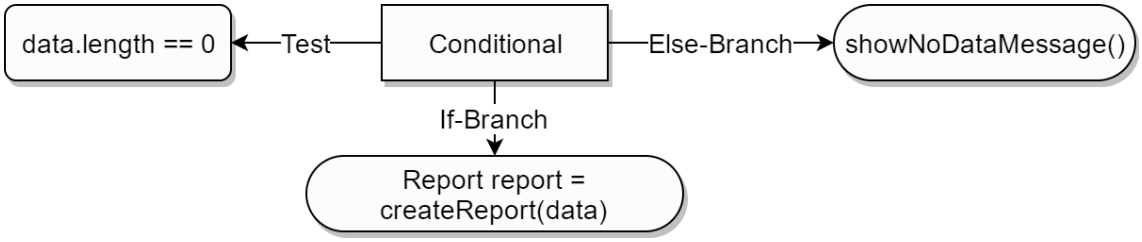

Рисунок 22 – условный оператор

Для реализации циклов используется оператор ForLoop, представляющий собой узел, имеющий четыре именованных выхода (см. рисунок 23):

- действие Setup – выражение инициализации индексатора цикла For,
- действие Test – условие останова цикла,
- действие Step – шаг индексатора цикла,
- Body – действие либо последовательность действий, представляющих тело цикла.

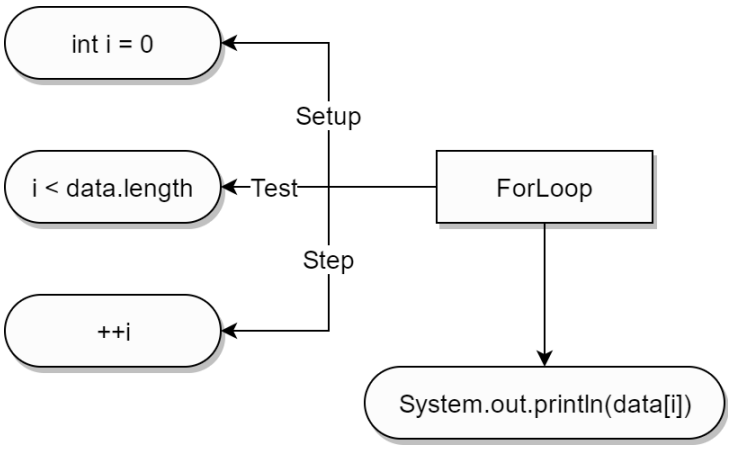

Рисунок 23 – цикл For

На рисунке 24 изображено промежуточное представление модели программы, выводящей данные из массива строк на стандартный вывод.

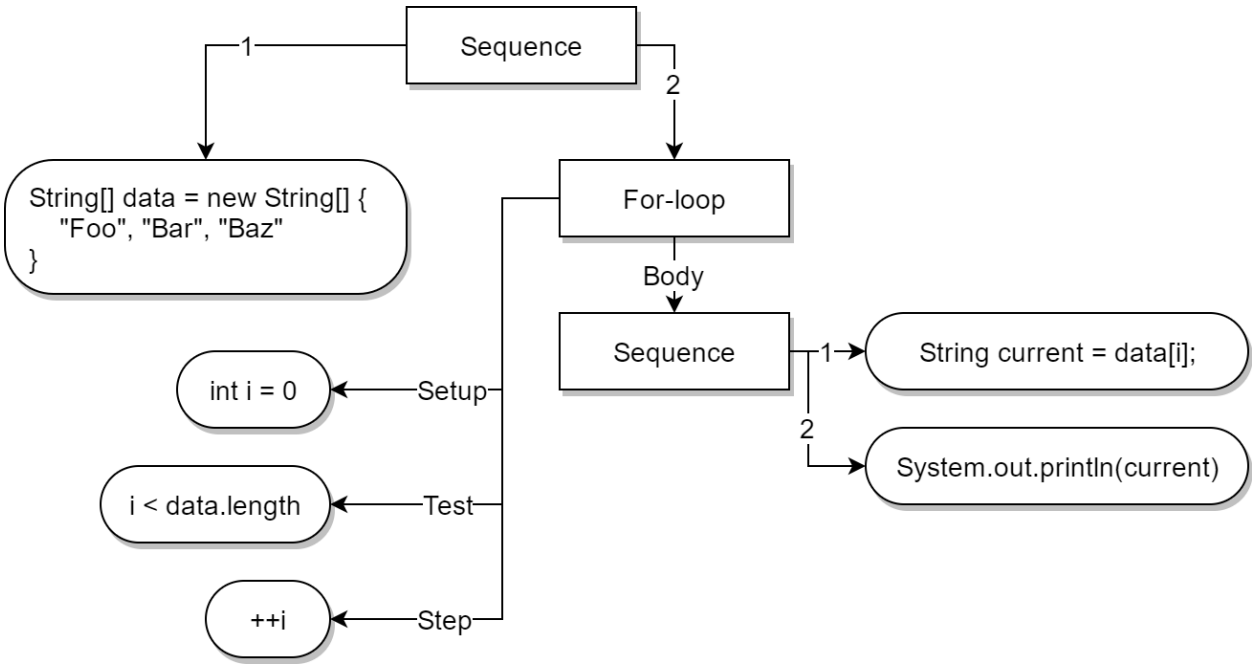

Рисунок 24 – графическое представление промежуточной модели

На рисунке 25 представлен пример мультиверсинного программного обеспечения.

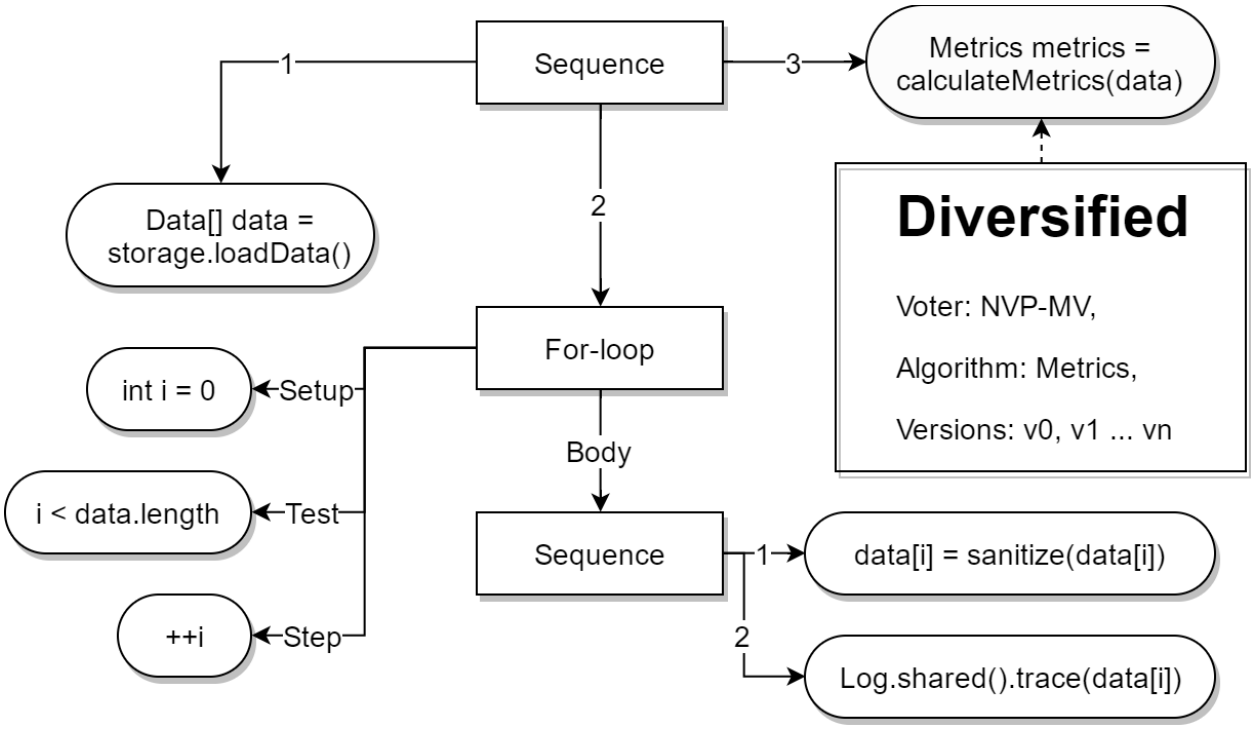

Рисунок 25 – графическое представление промежуточной модели

4.3. Процесс кодогенерации

Процесс кодогенерация – последовательное преобразование исходной модели в программный код на целевом языке программирования.

Процесс кодогенерации можно разделить на следующие этапы:

- Преобразование исходной модели в промежуточную,
- Преобразование промежуточной модели в программный код.

Правила преобразования диаграммы деятельности в промежуточную модель:

- Действия и узлы деятельности UML диаграммы преобразуются в действия промежуточной модели,
- Для последовательно соединённых действий и узлов деятельности создаётся один общий предок (узел последовательности), связи между ними разрываются. Пример такого преобразования представлен на рисунке 26,
- Ветвление в исходной модели заменяется на условный оператор промежуточной модели (см. рисунок 27-28),
- Циклы в исходной модели заменяются на соответствующий оператор цикла промежуточной модели (см. рисунок 29-30),
- Диверсифицированные действия исходной модели преобразуются в аналогичный узел промежуточной модели с сохранением параметров диверсификации (контекста).

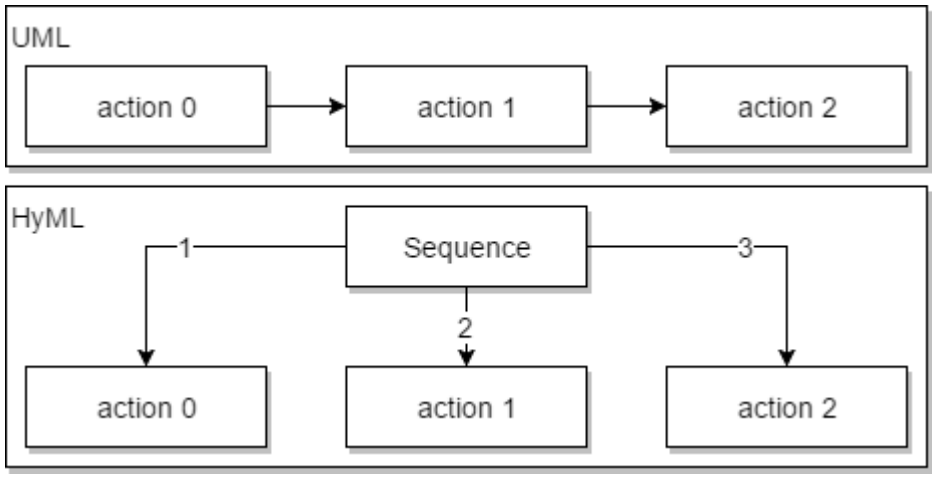

Рисунок 26 – последовательность действий

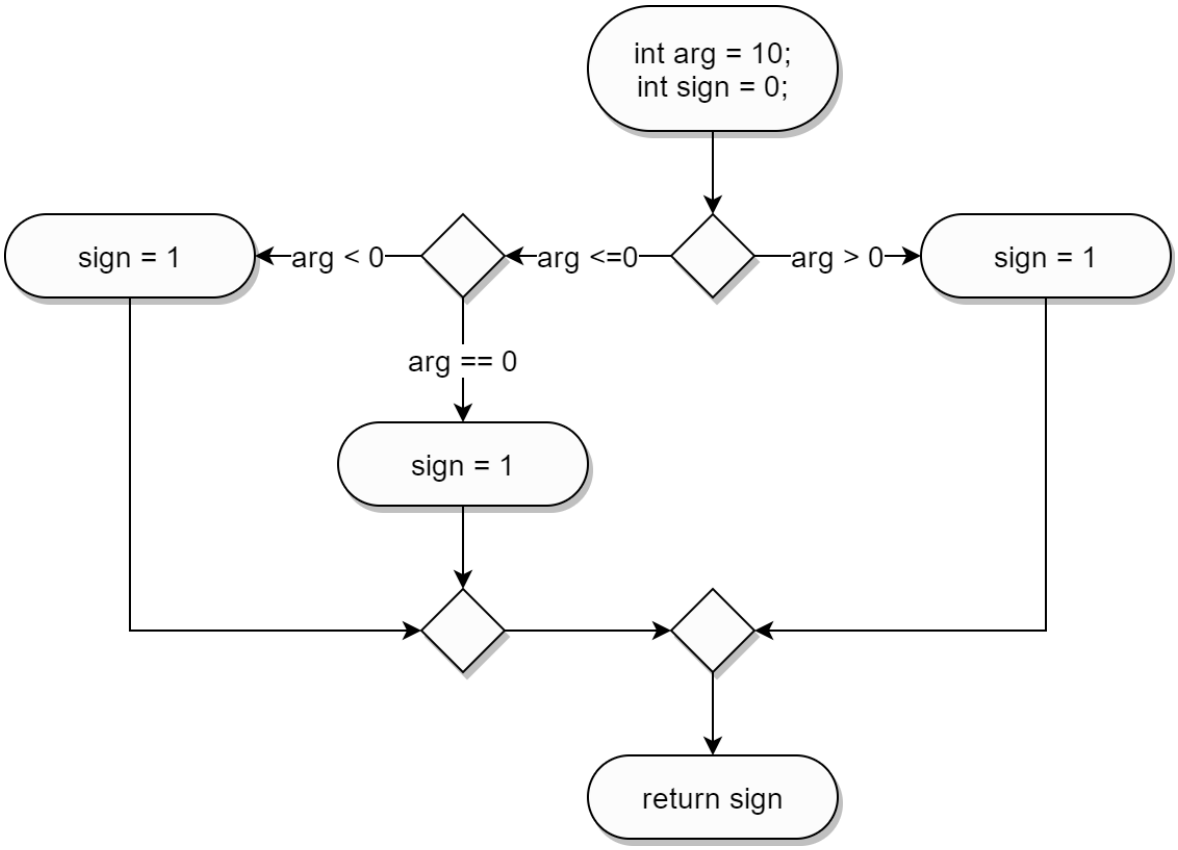

Рисунок 27 – представление условного оператора в UML

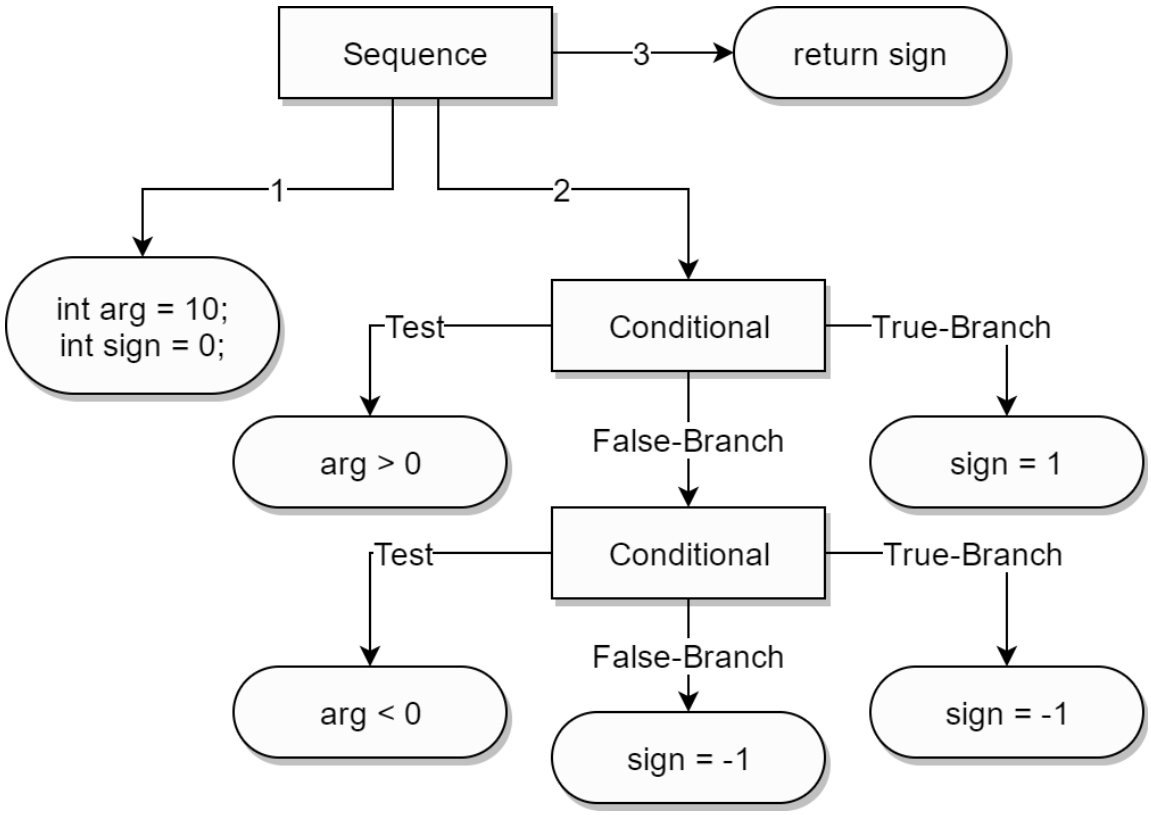

Рисунок 28 – представление условного оператора в промежуточной модели

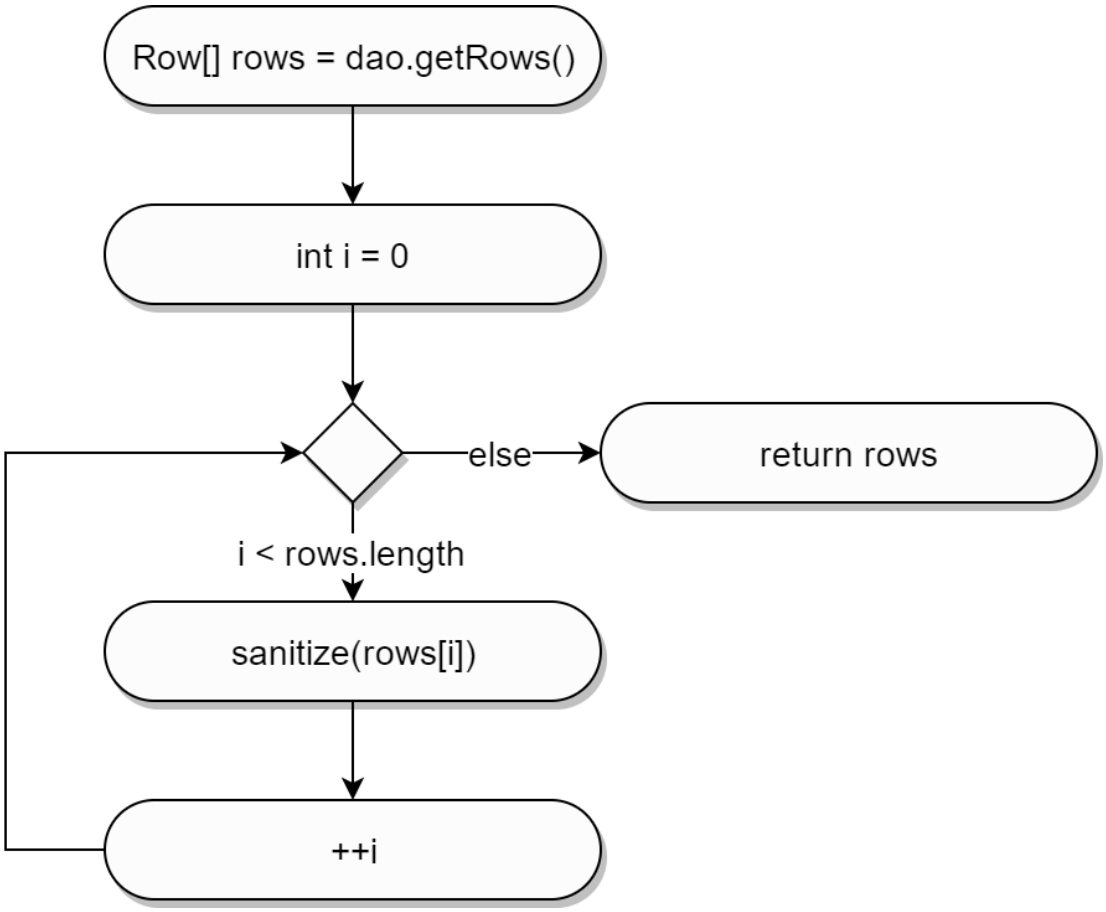

Рисунок 29 – представление цикла For в UML

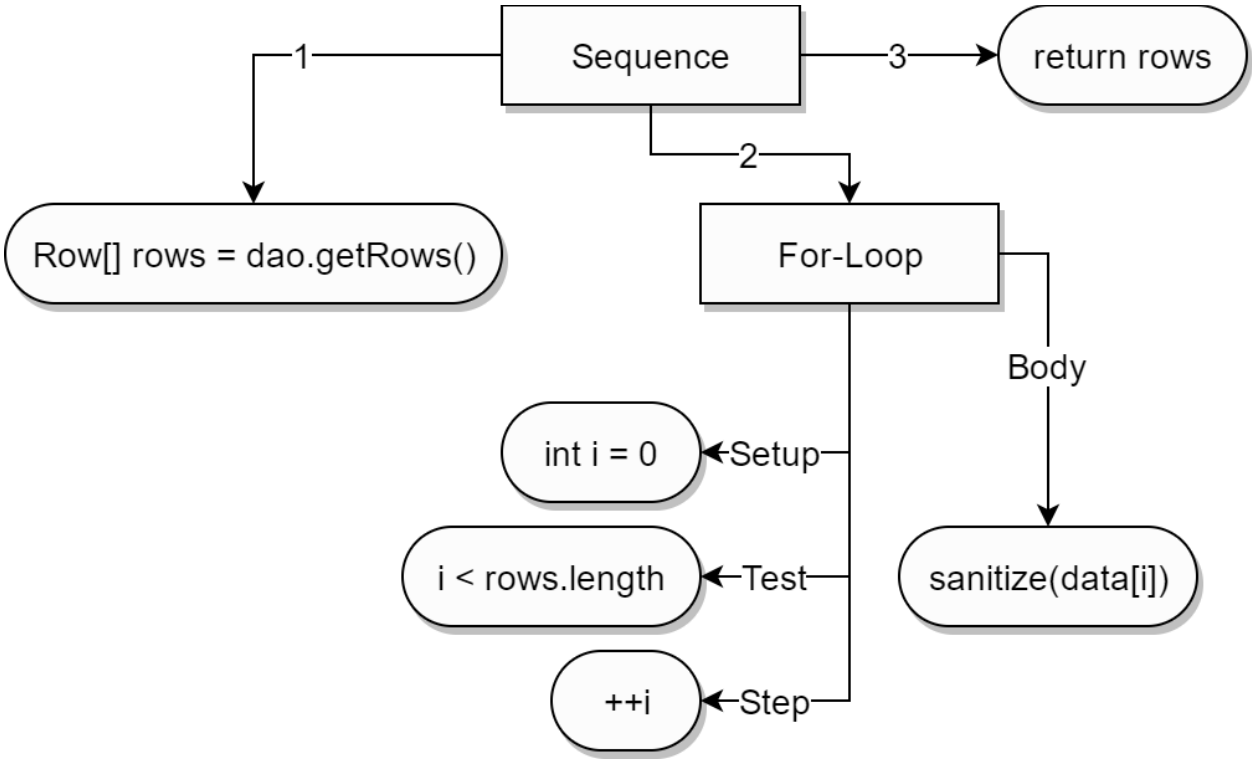

Рисунок 30 – представление цикла For в промежуточной модели

Правила преобразования промежуточной в Java-код:

Действия. Так как узлы действия содержат исходный код на целевом языке программирования в качестве своего значения, то при преобразовании из промежуточной модели используются их значения без изменений. За исключение случаев, когда значение узла действия завершает символом точки с запятой. В таком случае символ точки с запятой игнорирует, а оставшаяся часть значения копируется без изменения.

Последовательность. Для узлов, входящих в последовательность, генерируется исходный код согласно соответствующим их типу правилам. Между каждым выражением ставится символ точки с запятой, а вся последовательность окружается фигурными скобками.

Условный оператор. Условный оператор промежуточной модели представляет условный оператор целевого языка: if (*test*) *true-branch* [else *false-branch*]. Действуют следующие правила:

- Блоку *Test* узла промежуточно модели соответствует условное выражение *test*,
- Блоку *If-Branch* соответствует выражение либо последовательность выражений *true-branch*,
- Блоку *Else-Branch* соответствует выражение либо последовательность выражений *false-branch.*

Оператор цикла For промежуточной модели представляет соответствующий оператор целевого языка программирования: for (*setup*; *test*; *step*) *body*. Действуют следующие правила:

- Блоку *Setup* соответствует выражение инициализации индексатора цикла *setup*,
- Блоку *Test* соответствует логическое выражение (условие останова) *test*,
- Блоку *Step* соответствует выражение изменения индексатора цикла на каждом шаге *step*,

- Блоку *Body* соответствует выражение либо последовательность выражений (тело цикла) *body*.

Диверсифицированный вызов преобразуется в выражения вида: *Type Symbol = Function ‘(‘ [Argument ‘,’]* Argument ‘)’*. Кроме этого добавляется метод, в котором непосредственно происходит вызов диверсифицированного алгоритма. Диверсифицированный метод составляется с применением разработанного ранее микрофреймворка. Пример такого метода представлен ниже:

```
private T algorithm(Argument arg0, Argument arg1) {
    ActorSystem system = ActorSystem.create("random");
    GenVersion<T> v0 = TypedActor.get(system).typedActorOf(
        new TypedProps<>(GenVersion.class,
            () -> new Version0(arg0, arg1)));
    GenVersion<T> v1 = TypedActor.get(system).typedActorOf(
        new TypedProps<>(GenVersion.class,
            () -> new Version1(arg0, arg1)));
    GenVersion<T> v2 = TypedActor.get(system).typedActorOf(
        new TypedProps<>(GenVersion.class,
            () -> new Version2(arg0, arg1)));
    T res = voter.vote(Arrays.asList(v0, v1, v2)).get();
    system.terminate();
    return res;
}
```

Диверсифицированный метод имеет следующую сигнатуру:

- Тип возврата *T* выбирается на основе заданных параметров диверсификации. Равен типу возврата используемого диверсифицируемого алгоритма,
- Аргументы, соответствующие выбранному алгоритму.

4.4. Вывод

В данной главе была рассмотрена кодогенерационная модель диверсифицированного Java-кода из UML диаграмм деятельности.

Разработанная промежуточная модель позволяет абстрагироваться от специфики исходной модели, что даёт возможность использовать другие виды исходных моделей без необходимости изменения самой процедуры кодогенерации.

5. Разработка инструментария

Разработанный инструментарий Hydra представляет собой упрощенный редактор UML диаграмм деятельности, позволяющий генерировать программный код на языке Java 8.

При разработке инструментария были применены следующие технологии:

- Платформа: Java,
- Языки программирования: Java 8 и Scala 2.11 (только стандартная библиотека scala-lib),
- Java Swing GUI – библиотека для разработки графических приложений,
- Akka – библиотека реализующая модель акторов,
- RxJava – библиотека, реализующая основные концепты реактивного программирования,
- JGraphX – компонент графа,
- RSyntaxTextArea – компонент текстового редактора с подсветкой синтаксиса,
- WebLaF – дополнение к Java Swing, представляющее собой набор улучшенных Swing-компонентов, а также Look and Feel (стили оформления),
- шрифты Awesome Font и адаптер для Java jIconFont,
- Система сборки Maven.

Инструментарий разбит на несколько модулей:

- Микрофреймворк Zmok, предоставляющий абстракции над основными блоками NVP, а также содержащий реализации алгоритмов голосования,
- Модуль Hydra.Utils – набор классов-утилит,
- Модуль Hydra.Core – ядро системы. Содержит все классы моделей, парсера языка выражений и трансформаторов,
- Модуль Hydra.App – редактор.

Компоненты инструментария и их зависимости изображены на рисунке 31.

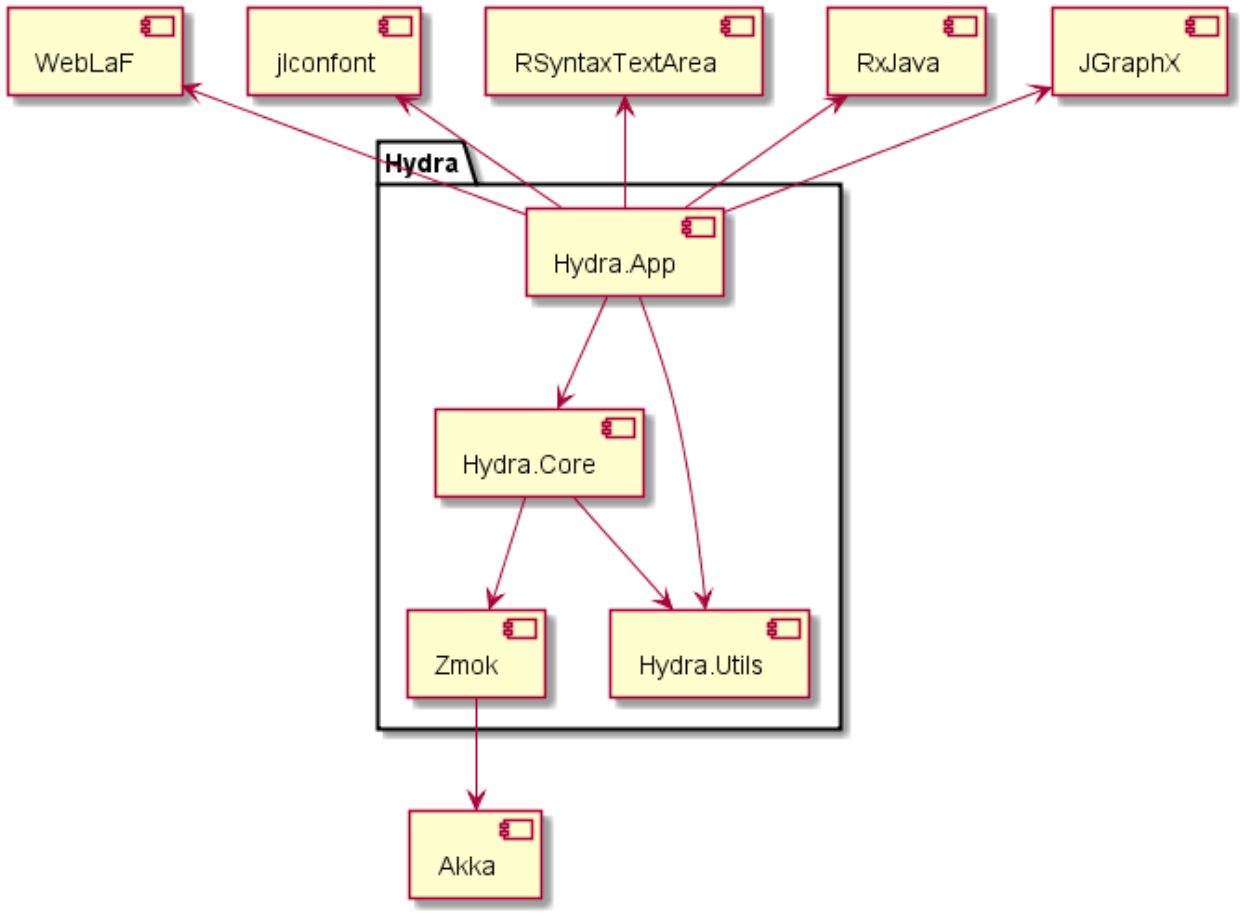

Рисунок 31 – граф зависимостей ПО

5.1. Микрофреймворк Zmok

Микрофреймворк Zmok основан на библиотеке Akka, реализующей модель акторов [23].

Разработанный фреймворк представляет собой набор абстракций над основными блоками NVP:

Обобщённая версия (*GenVersion[T])* представляет версию программного модуля, возвращающую результат типа T. Интерфейс *GenVersion* предоставляет следующие методы: *getHeuristic :: () → Option[T]* и *getHeuristicAsync :: () → Future[T]*, где *Option* и *Future* – соответствующие монады.

Монада – специальный контейнер, позволяющий абстрагироваться от вычислений [24]. Например, монада Maybe (Option) предоставляет абстракцию над результатом вычислений (вместо результата может быть возвращена ошибка), а монада Future представляет результат вычислений, который будет получен позже (то есть вычисления производятся асинхронно).

Обобщённый алгоритм голосования (*GenVoter[T]*) представляет алгоритм голосования, выбирающий решение из результатов версий, типизированных T.

Классы *MajorityVoter* и *ConsensusVoter* реализуют алгоритмы голосования абсолютным (NVP-MV) и согласованным большинством (NVP-CV) соответственно.

Классы пакета matrix используются исключительно в служебных целях (в реализованных алгоритмах голосования).

Диаграмма классов микрофреймворка представлена на рисунке 32.

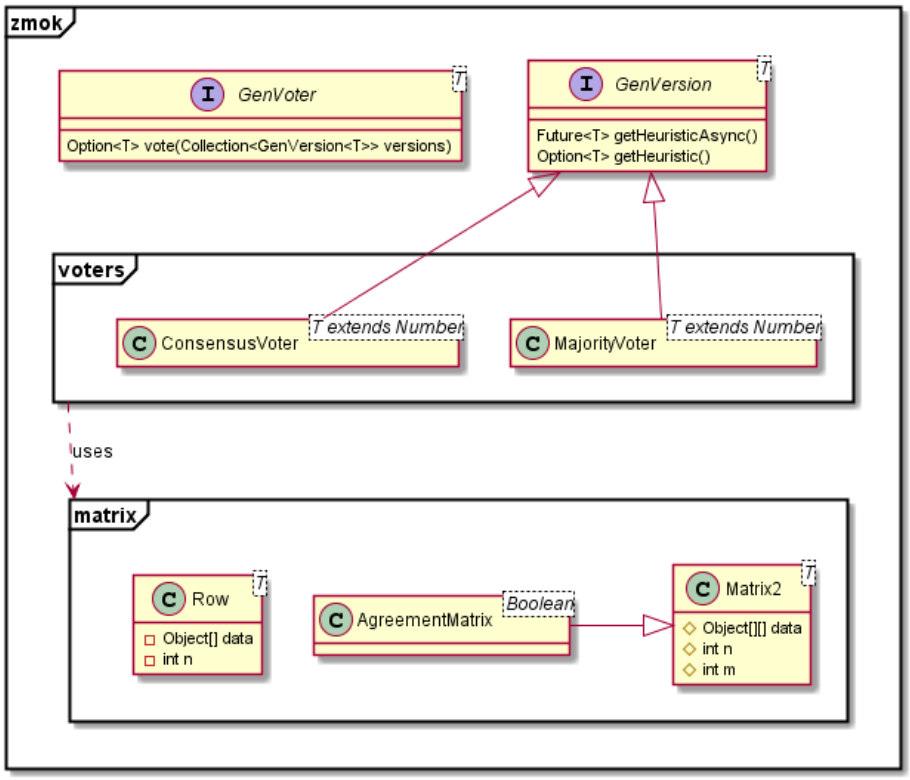

Рисунок 32 – диаграмма классов микрофреймворка

Пример использования микрофреймворка:

Реализация версии:

```
public class GetNumber implements GenVersion<Integer> {
    private int value;
    public GetNumber(int value) { this.value = value; }
    @Override Option<Integer> getHeuristic() {
        return Option.option(value);
    }
    @Override Future<Integer> getHeuristicAsync() {
        return Futures.successful(value);
    }
}
```

Создание акторов версий и алгоритма голосования, вызов и получение результата:

```
ActorSystem system = ActorSystem.create("getNumber"); // (1)
GenVersion<Integer> v0 = TypedActor.get(system)
    .typedActorOf(new TypedProps<>(
        GenVersion.class,
```

```
            () -> new GetNumber(4))); // (2)
    ...
    GenVoter<Integer> voter = TypedActor.get(system)
        .typedActorOf(new TypedProps<>(
            GenVoter.class,
            MajorityVoter.class)); // (3)
    Option<Integer> result = voter
        .vote(Arrays.asList(v0, ...); // (4)
    system.terminate(); // (5)
```

(1) создание системы акторов с помощью Akka,

(2) создание актора версии GetNumber с аргументом «4»,

(3) создание актора алгоритма голосования абсолютным большинством (MajorityVoter)

(4) вызов версий и алгоритма голосования,

(5) закрытие системы акторов.

5.2. Классы утилит, модуль Hydra.Utils

Модуль *Hydra.Utils* содержит всевозможные классы-утилиты, а именно:

- *Collections* – предикаты над элементами списка, конвертация из массива в список (в том числе null-безопасная) и обратно,
- *Contracts* – контракты,
- *Dictionary* – null-безопасная реализация ассоциативного массива (*HashMap*),
- *Strings* – методы для конструирования строк,
- *Tuple* – кортеж из двух элементов,
- аннотация *ImplicitUsage* – аннотация, сигнализирующая о неявном использовании сущности,
- интерфейс *Initializable* – предоставляет интерфейс для инициализируемых компонентов,

- классы-форматировщики *Parenthesised* (окружённый круглыми скобками), *Quoted* (окружённый двойными кавычками),
- функторы *Action* :: *T* → *Void*, *Action2 :: (T, U)* → *Void*, *Proc :: Void*,
- пакет *io.xml*: набор классов для работы со внешними XML-документами, сериализуемых с помощью JAXB.

5.3. Ядро системы, модуль Hydra.Core

Ядро системы можно разбить на следующие группы:

- Модели данных: обобщённая модель графа, диаграмма деятельности, промежуточная модель, внутреннее представление Java-кода,
- Модули диверсификации: модель данных банка алгоритмов, классы работы с XML-документом банка алгоритмов,
- Парсер языка выражений диверсифицированных вызовов,
- Классы-преобразователи, реализующие описанную выше кодогенерационную модель.

5.3.1. Обобщённая модель графа

Обобщённая модель графа состоит из узлов, соединённых между собой рёбрами. Диаграмма классов представлена на рисунке 33.

Класс *Node* представляет абстрактный узел, содержащий некоторое типизированное значение *value :: T*. Для идентификации узлов используется неизменяемое поле *id :: long*.

Класс *Edge* представляет абстрактную связь между узлами типа *TNode*. Каждое соединение имеет: уникальный идентификатор *id :: long*, *источник src :: TNode* и приемник *dst :: TNode*, направление *direction :: EdgeDirection*, а также тип *type :: EdgeType*.

Перечисление *EdgeDirection* представляет направление ребра графа. Может принимать одно из следующих значений: *OneWay* (односторонняя связь от источника к приемнику), *OneWayToSource* (одностороння связь от приемника к источнику), *TwoWay* (двунаправленная связь).

Перечисление *EdgeFlowDirection* используется исключительно для служебных целей – поиска рёбер узла определённого типа.

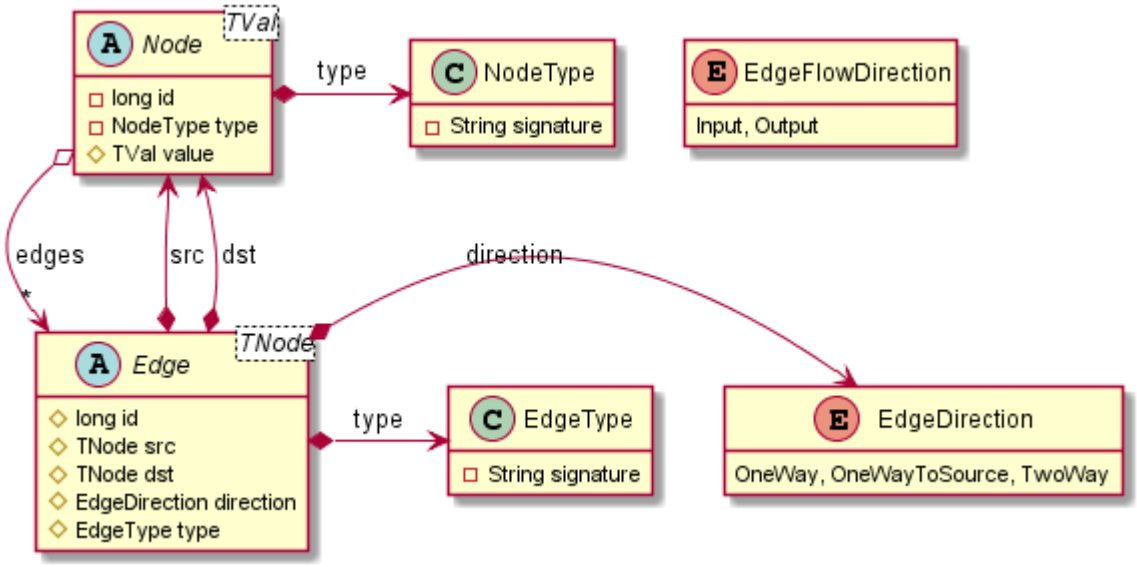

Рисунок 33 – диаграмма классов обобщённой модели графа

5.3.2. Модель диаграммы деятельности

В разработанной программной системе реализована модель диаграммы деятельности со следующими ограничениями:

- Узлы деятельности совмещены с действиями,
- Отсутствуют элементы управления параллелизмом: fork и join,
- Отсутствуют «плавательные дорожки»,
- Отсутствуют области расширения,
- Вместо узлов организации ветвления используются структурированные действия: условный оператор *Conditional* и цикл с параметрами *For*,
- Структурированное действие *Conditional* представляет условный оператор и имеет следующую структуру: условие *Test*; действия, выполняющиеся при истинности условия, *If-Branch*; действия, выполняющиеся при ложности условия, *Else-Branch*,

- Структурированное действие *For* представляет цикл с параметрами и имеет следующую структуру: инициализация индексатора цикла *Setup*, условия останова *Test*, шаг индексатора цикла *Step* и тело цикла *Body*.

Модель диаграммы деятельности строится на основе обобщённой модели графа, где каждый узел типизирован строковой (*String*). На рисунках 33 и 34 изображена базовая часть модели, на рисунке 35 – узлы диаграммы деятельности.

Класс *UmlGraph* предоставляет интерфейс для создания и хранения узлов графа и рёбер между ними,

Класс *UmlNode* – базовый класс узлов диаграммы деятельности. Имеет UML-стереотип Stereotype и спецификацию поведения *UmlNodeSpecification*,

Класс *UmlEdge* – базовый класс для рёбер диаграммы деятельности,

Класс *UmlEdgeType* представляет тип ребра, а также выступает в роли фабрики – позволяет создавать узлы данного типа. Все объекты типа *UmlEdgeType* являются синглтонами и описаны в классе *UmlEdgeTypes*.

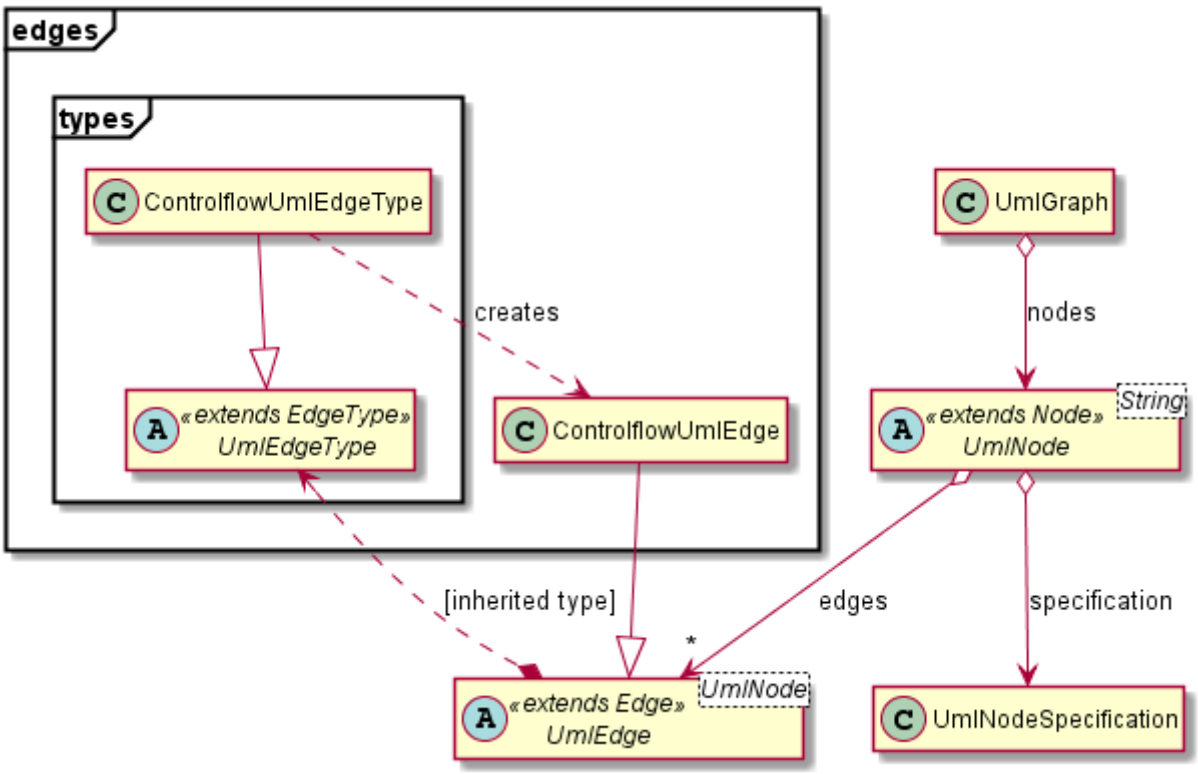

Рисунок 34 – базовая часть модели диаграммы деятельности

В текущей версии инструментария реализован один тип связей, а именно поток управления (controlflow).

Связь типа «поток управления» представлена классами *ControlflowUmlEdge* и *ControlflowUmlEdgeType*. Связь данного типа имеет один необязательный параметр: условие перехода.

Пакет specification содержит набор классов, определяющих поведение узлов диаграммы. На данный момент реализован и используется только один тип поведения, определяющий мощность соединения того или иного типа,

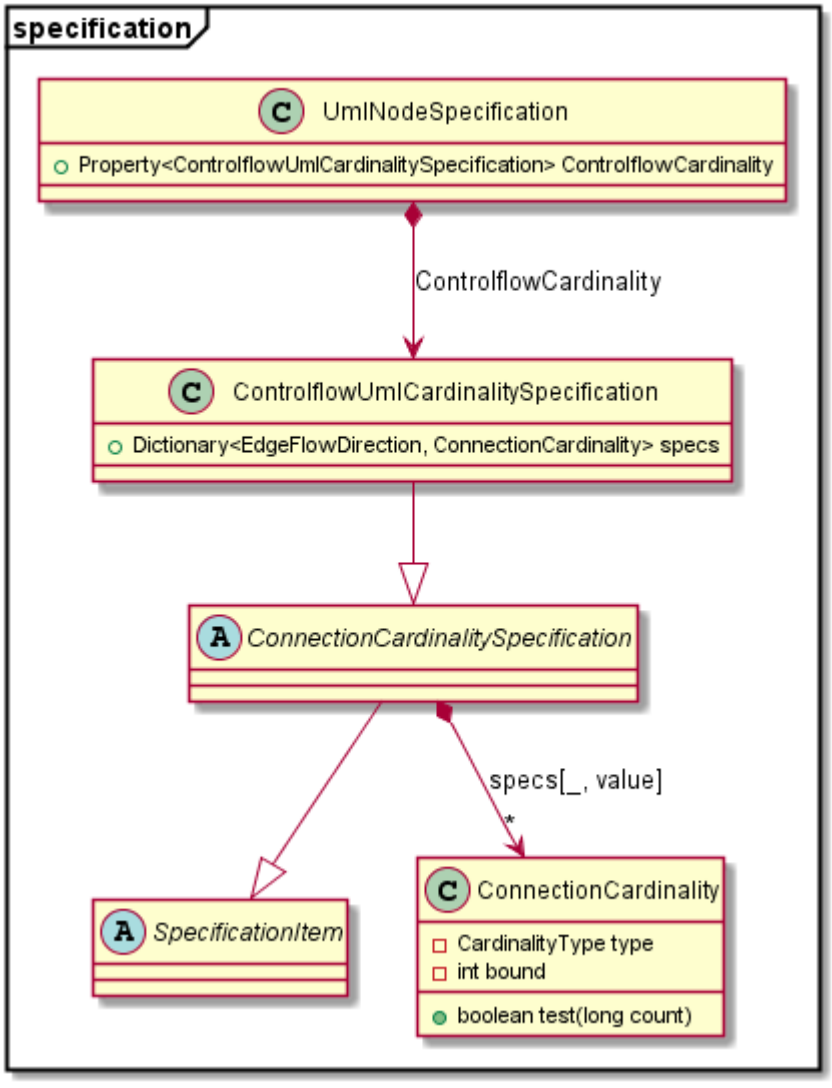

Рисунок 35 – базовая часть модели диаграммы деятельности

На рисунке 36 изображена иерархия классов узлов диаграммы деятельности. Для удобства отображения стереотипы узлов используются в качестве списка реализованных интерфейсов.

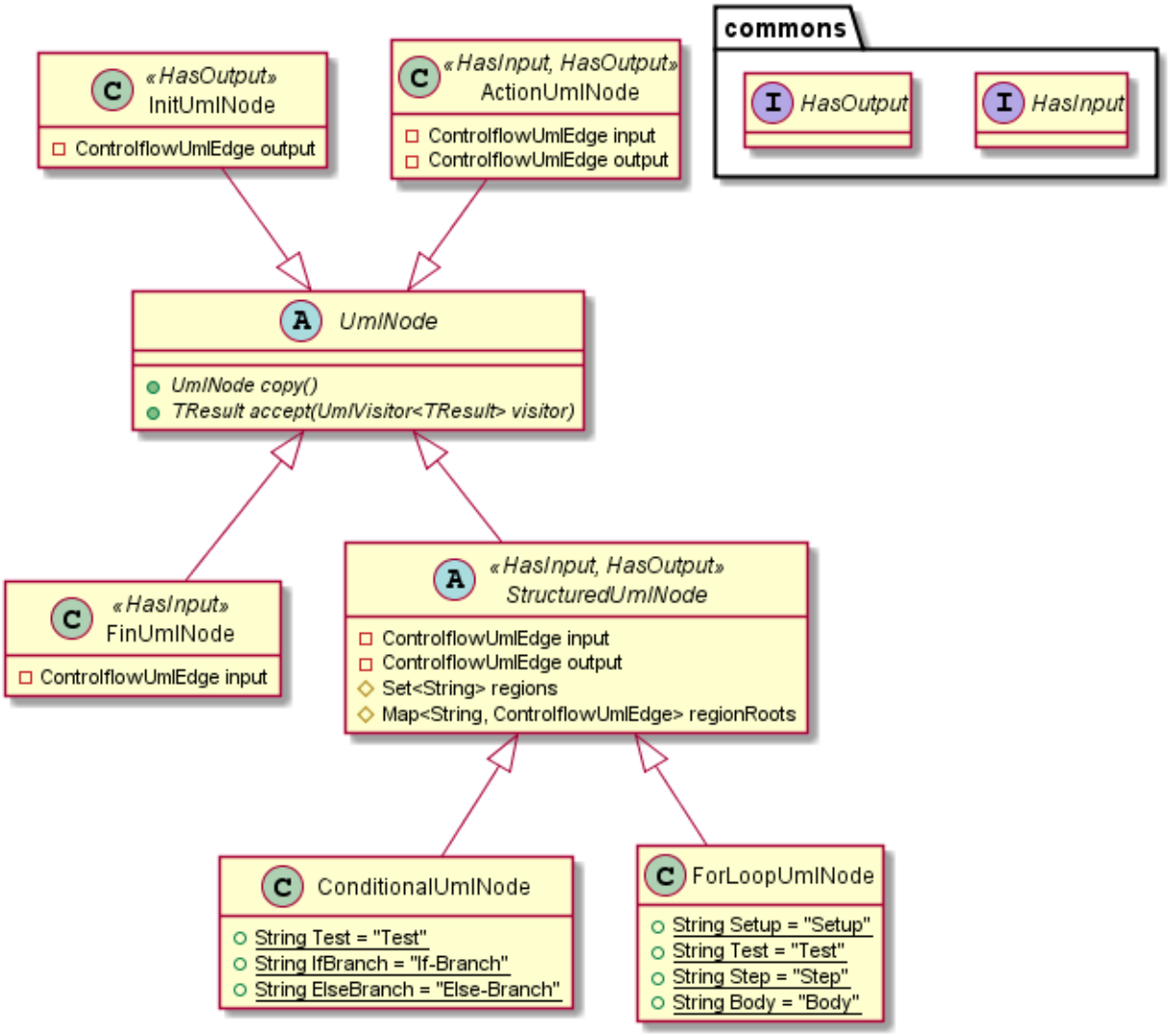

Рисунок 36 – узлы модели диаграммы деятельности

5.3.3. Промежуточная модель

Промежуточная модель представлена в виде ацикличного ориентированного графа (дерева). Диаграмма классов изображена на рисунке 37.

Модель состоит из следующих классов:

- Класс *HyNode* – базовый класс для всех узлов промежуточной модели,
- Класс *HySequence* – представляет последовательность узлов промежуточной модели,
- Класс *HyAction* – представляет выражение на целевом языке программирования либо последовательность выражений, перечисленных через символ точки с запятой,
- Класс *HyDiversifiedAction* – представляет диверсифицированный вызов. Содержит информацию о параметрах диверсификации (контекст), которые полностью соответствую стереотипу исходного узла UML диаграммы деятельности,
- Класс *HyConditional* представляет условный оператор и имеет следующую структуру: условие *Test*; действия, выполняющиеся при истинности условия, *If-Branch*; действия, выполняющиеся при ложности условия, *Else-Branch*,
- Класс *HyForLoop* – представляет цикл с параметром и имеет следующую структуру: инициализация индексатора цикла *Setup*, условия останова *Test*, шаг индексатора цикла *Step* и тело цикла *Body*.

Помимо самих классов, представляющих узлы промежуточной модели, присутствует один дополнительный, – *HyVisitor*, – реализующий шаблон «посетитель» (visitor). Шаблон Visitor позволяет вынести обработки всех узлов модели в отдельный класс. Преимуществом такого подхода является его расширяемость [25].

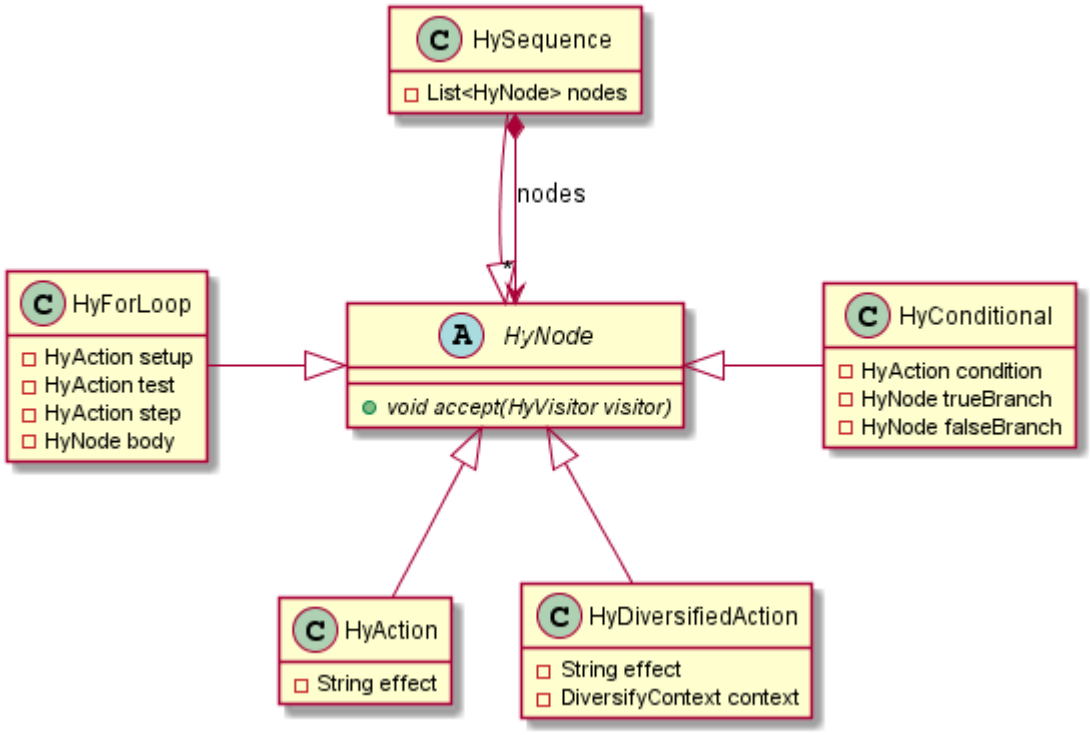

Рисунок 37 – диаграмма классов промежуточной модели

5.3.4. Модуль диверсификации: банк алгоритмов и контекст диверсификации

Банк алгоритмов содержит описание всех алгоритмов и версий, доступных в системе. Хранится в виде XML-документа со структурой, изображённой на рисунке 37 (документ строится автоматически на основе его объектного представления с помощью JAXB).

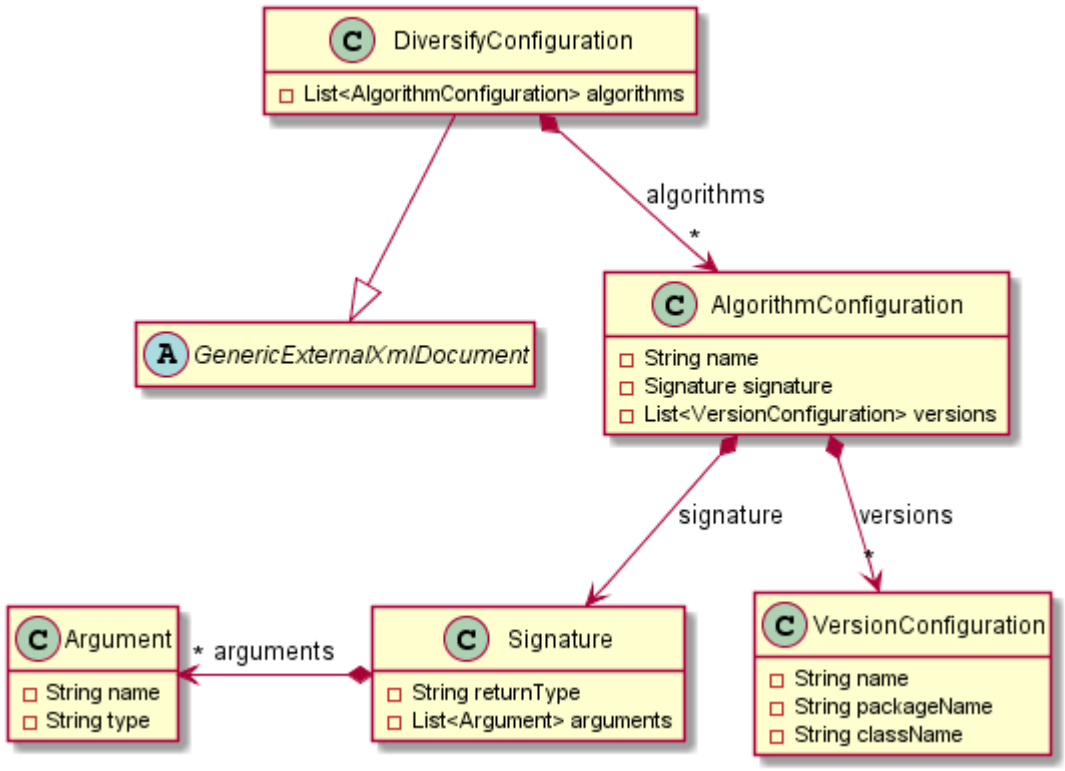

Рисунок 38 – диаграмма классов Xml-представления банка алгоритмов

Класс *DiversificationContext* представляет контекст диверсификации, имеющий следующую структуру:

- Используемый алгоритм голосования,
- Диверсифицируемый алгоритм,
- Набор версий диверсифицированного алгоритма.

5.4. Редактор

Редактор диаграмм разработан с применением следующих технологий: графическая библиотека Java Swing, компонент графа JGraphX, система сообщений RxJava, компонент текстового редактора с подсветкой синтаксиса RSyntaxTextArea. Главное окно программы изображено на рисунке 39.

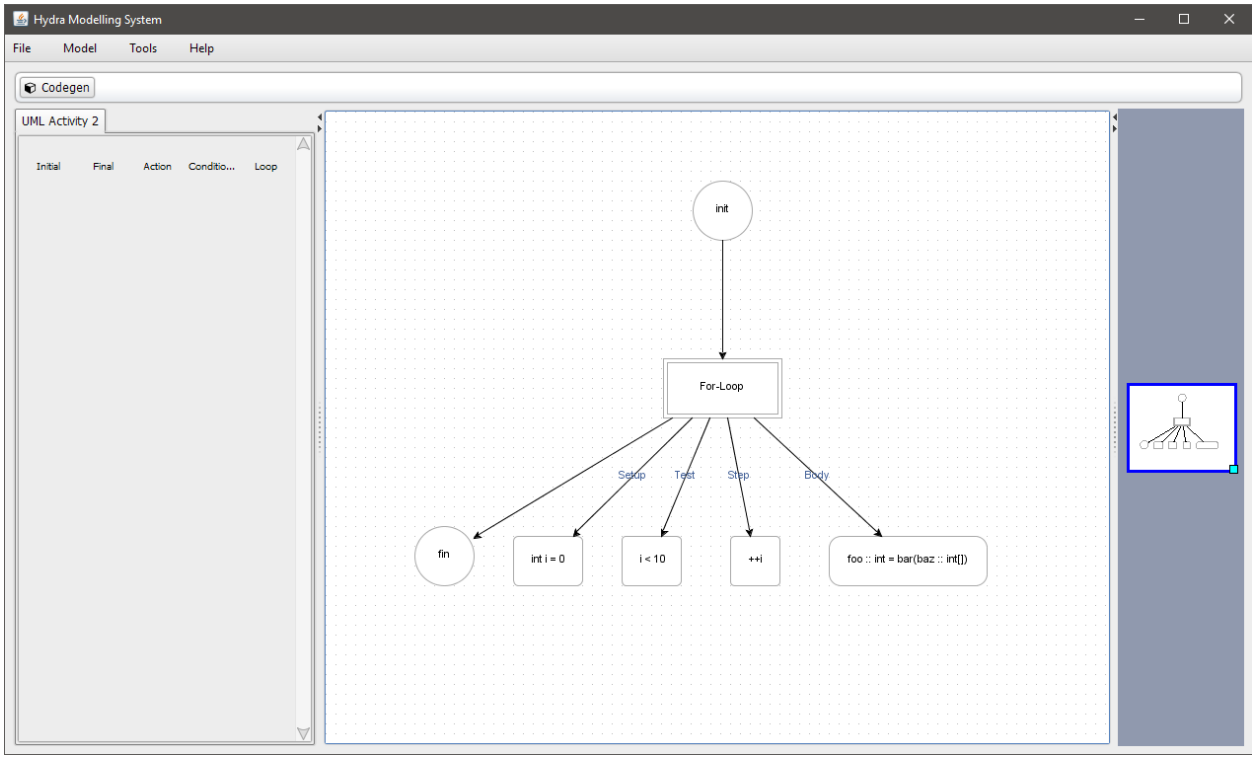

Рисунок 39 – главное окно программы

5.4.1. Система событий

Система событий в разработанном инструментарии основана на библиотеке RxJava, реализующей элементы реактивного программирования.

Основными элементами системы событий, построенной по методологии реактивного программирования, являются «издатели» (publishers, в терминологии RxJava – observables), играющие роль источников данных, и «подписчики» или «наблюдатели» (subscribers, observers), объекты принимающие события.

Объекты-наблюдатели подписываются на источник данных. Затем эти наблюдатели реагируют на события, генерируемые источником.

Каждый наблюдатель должен реализовывать следующие методы:

- *onNext* – обработка события, сгенерированного источников,
- *onError* – обработка ошибки, произошедшей в источнике,
- *onCompleted* – обработка события завершения работы источника.

Источник данных соблюдает следующий контракт работы с этими методами:

- Метод *onNext* может быть вызван нуль и более раз,
- Методы *onError* и *onCompleted* могут быть вызваны только раз,
- После вызова метода *onError* или *onCompleted* время жизни источника оканчивается, и он более не генерирует сообщения.

Другим не менее важным элементом RxJava является класс *Subject* и его наследники. *Subject* представляет источника событий и наблюдателя одновременно. В RxJava реализована четыре вида *Subject'ов*:

PublishSubject. Источник данных пересылает данные наблюдателям мгновенно – сразу же при их получении. Принцип работы *PublishSubject* изображён на рисунке 40.

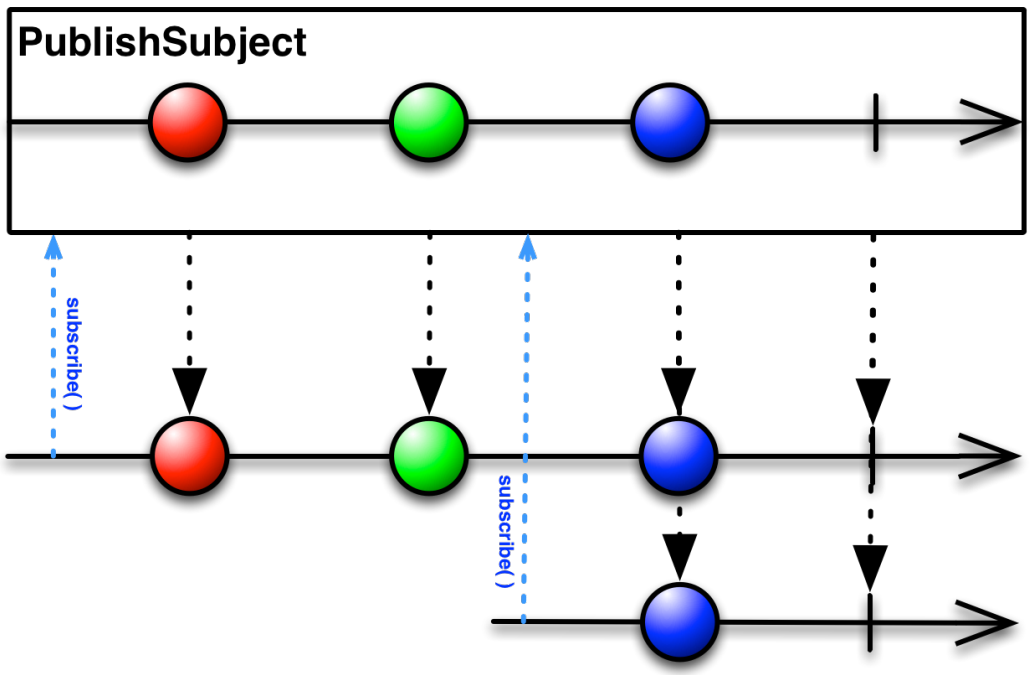

Рисунок 40 – принцип работы PublishSubscribe

AsyncSubject выполняет роль асинхронной задачи. Отправляет только последнее полученное сообщение по вызову *onCompleted*. Принцип работы *AsyncSubject* изображён на рисунке 41.

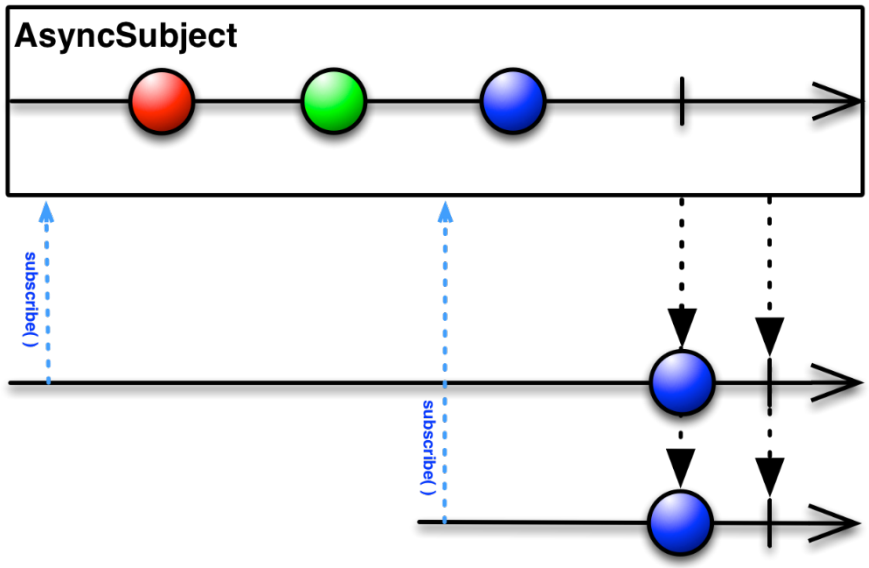

Рисунок 41 – принцип работы AsyncSubject

BehaviorSubject. При регистрации нового подписчика, *BehaviorSubject* пересылает ему своё последнее сообщение, далее продолжает работать в нормальном режиме. Принцип работы *BehaviorSubject* изображён на рисунке 42.

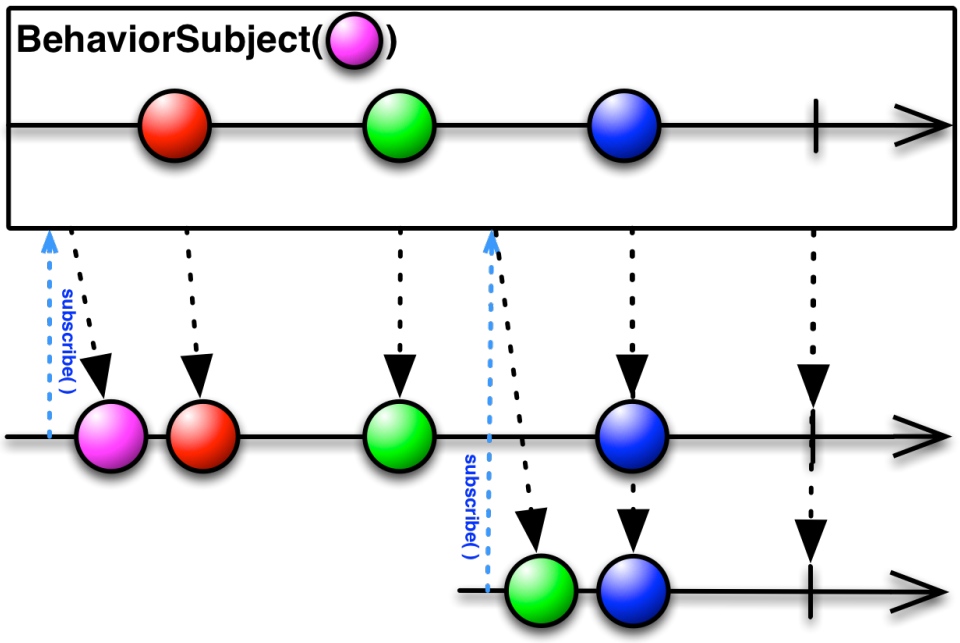

Рисунок 42 – принцип работы BehaviorSubject

ReplaySubject запоминает все полученные ранее данные и пересылает их всем зарегистрированным наблюдателям вне зависимости от времени их регистрации. Принцип работы *ReplaySubject* изображён на рисунке 43.

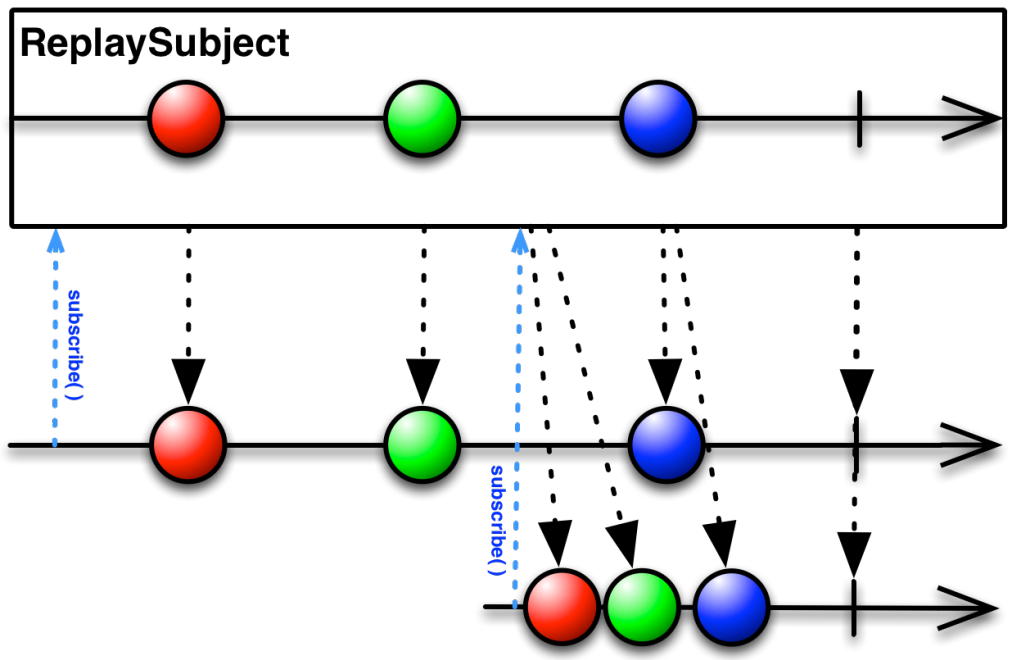

Рисунок 43 – принцип работы ReplaySubject

Для удобства работы с системой событий была реализована глобальная шина данных на основе PublishSubject. Шина представляет собой объект-синглтон, доступный из любой точки программы.

5.4.2. Модификация редактора графов

Редактор графов основан на библиотеке JGraphX. В ходе работы были модифицированы следующие элементы:

- Система описания стилей узлов графа,
- Компоненты mxGraph и mxGraphComponent.

Существенным недостатком встроенной в JGraphX системы описания стилей узлов графа является её «небезопасность». Полное описание стиля узла представляет собой строку вида:

$$(parameter=value;)^*$$

, где *parameter* и *value* – некоторые строки. Возможные имена параметров и их значения (если параметр имеет ограниченное число состояний) перечислены в отдельном статическом классе в виде константных полей. Такой подход является антипаттерном [26], так как не типобезопасен, то есть позволяет конструировать стили подобного вида: *rounded=SHAPE_RECTANGLE*, в то время как параметр rounded ожидает значение логического типа. Подобные несоответствия могут повлечь за собой создание исключения, а, следовательно, некорректное поведение программы. Для устранения данного недостатка была разработан набор классов, сделавших систему описания стилей типобезопасной. Диаграмма классов (сокращённая) представлена на рисунке 44.

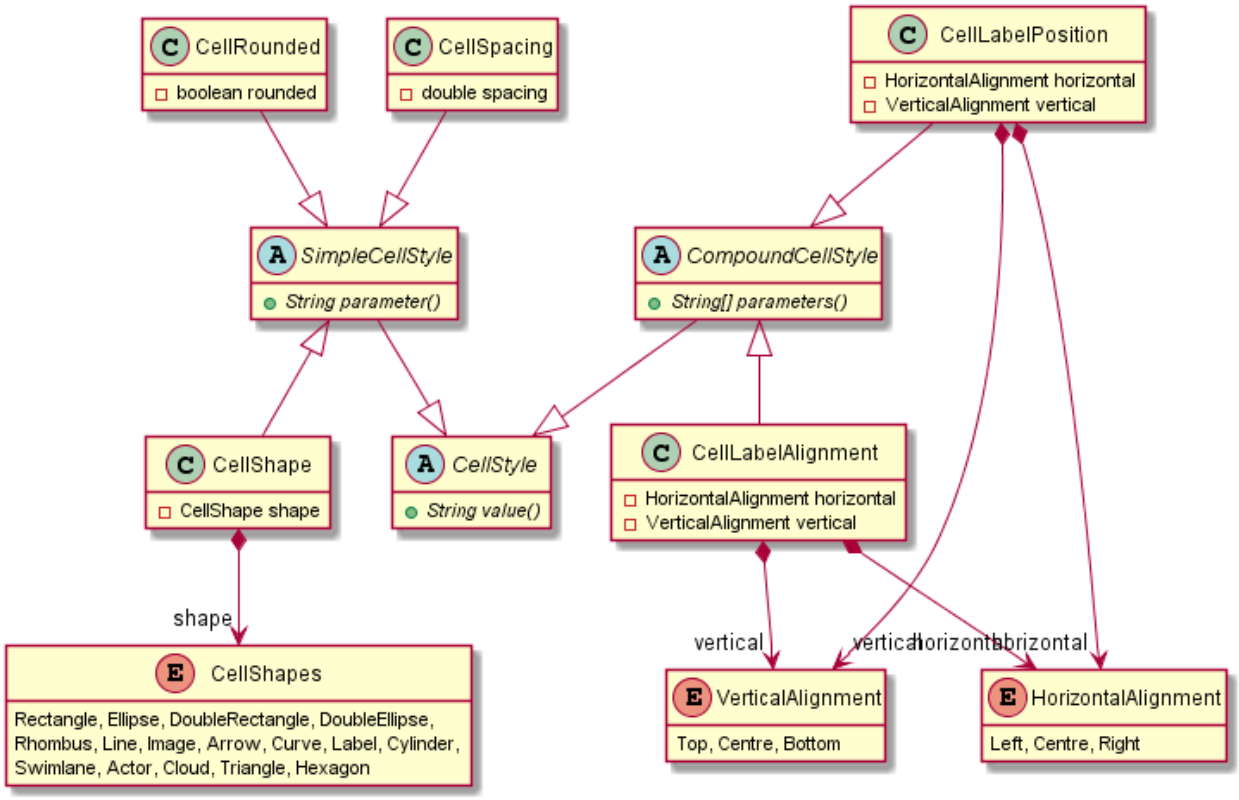

Рисунок 44 – диаграмма классов системы описания стилей

Таким образом описаны следующие параметры стилей узлов графа:

- *Foldable* – определяет можно ли сворачивать узел, имеет значение логического типа,

- *Overflow* – определяет параметры переноса текста в узле, может принимать одно из следующих значений: текст обрезается по размеру узла (*hidden*), текст может выходить за пределы узлы (*visible*), режим заполнения (*fill*),
- *Rounded* – позволяет скруглять углы у прямоугольных узлов, имеет значение логического типа,
- *Spacing* – определяет внутренние границы узла, имеет значение типа double,
- *Shape* – определяет форму узлы, может принимать одно из следующих значений: прямоугольник (*Rectangle*), эллипс (*Ellipse*), прямоугольник с двойными границами (*DoubleRectangle*), эллипс с двойными границами (*DoubleEllipse*), ромб (*Rhombus*) и т.д.,
- *LabelPosition* и *LabelAlign* – позиция и выравнивание текста в узле, состоят из двух параметров: позиция по вертикали и позиция по горизонтали. Для позиции по вертикале возможны следующие значения: сверху (*Top*), по центру (*Centre*), снизу (*Bottom*); для позиции по горизонтали: слева (*Left*), по центру (*Centre*), справа (*Right*).

Модификации классов были в mxGraph и mxGraphComponent произведены для изменения их стандартного поведения, а именно:

- Обработка событий соединения узлов: отображение изменений на реальную модель данных, вызов промежуточных окон,
- Обработка удаления узлов: отражение изменений на реальную модель данных,
- Проверка добавляемых соединений на корректность (согласно спецификации узла),
- Обработка добавления узлов из библиотеки компонентов.

Обработка события соединения узлов происходит по следующему алгоритму (см. рисунок 45):

```
ВХОД: созданное ребро edge
ЕСЛИ edge имеет тип UmlEdge,
ТО ВЫХОД // (1)
удалить edge из модели графа // (2)
ЕСЛИ edge.getSource() имеет тип RegionHeader,
ТО послать сообщение RegionConnectionEvent
   ВЫХОД // (3)
отобразить изменения на UML-модель
ВЫХОД
```

1) Если созданное ребро имеет тип UmlEdge, то такое ребро считается уже обработанным (т.е. событие было вызвано повторно),

2) Из-за особенностей реализации вызов события происходит дважды, поэтому первое, созданное автоматически, и необработанное ребро необходимо удалить,

3) При соединении структурного узла (Conditional или ForLoop) и любого другого пользователю предлагается выбрать к какому блоку будет принадлежать это соединение (например, инициализация цикла).

Алгоритм обработки события удаления соединений (см. рисунок 46):

```
ВХОД: список удалённых рёбер edges
ДЛЯ ВСЕХ i в edges:
  source = i.getSource().getUml()
  target = i.getTarget().getUml()
  ДЛЯ ВСЕХ j в source.edges():
    ЕСЛИ j.getTarget() == target,
    ТО edge.getType().remove(j)
ВЫХОД
```

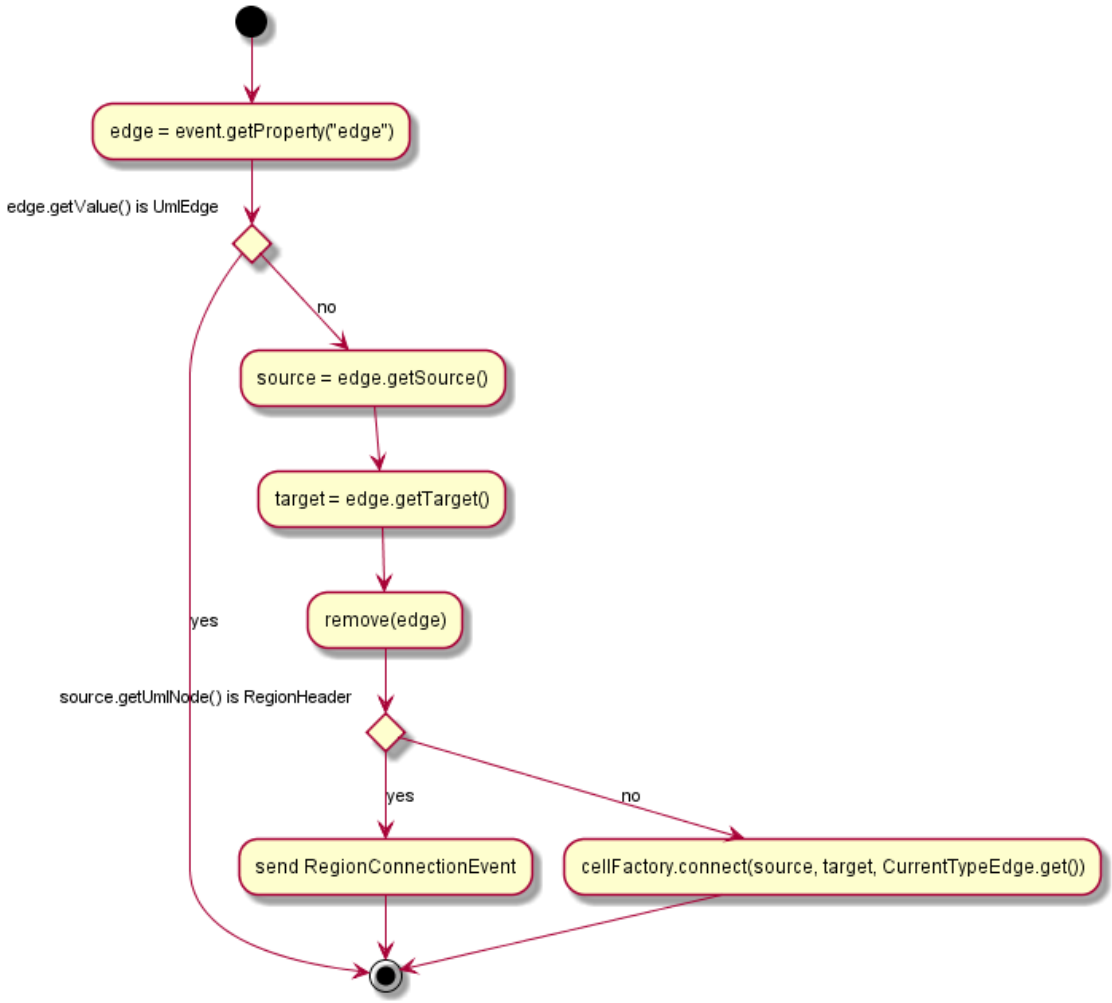

Рисунок 45 – алгоритм обработки создания ребра между узлами

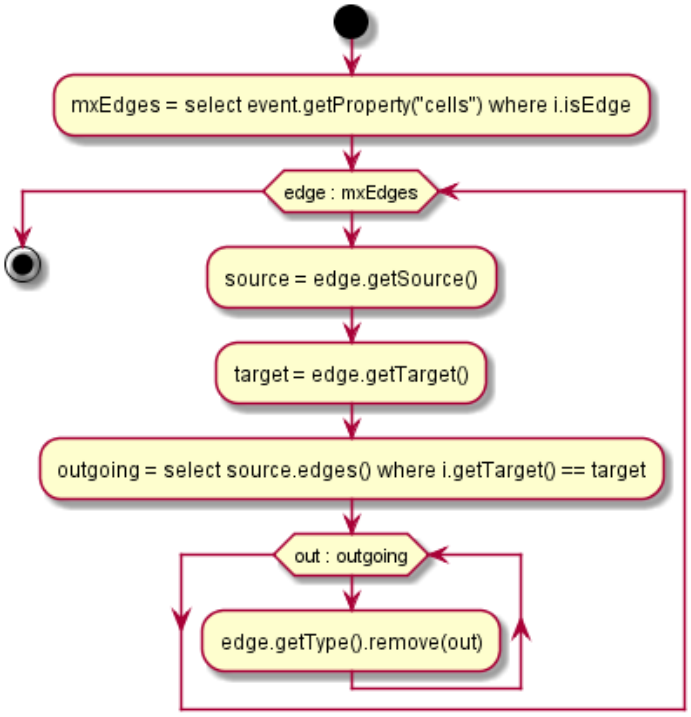

Рисунок 46 – алгоритм обработки удаления рёбер

5.4.3. Пример работы программы

Главное окно программы представлено на рисунке 47.

1) библиотека UML-элементов. Позволяет добавлять элементы на рабочую область путём перетягивания элемента из библиотеки на область графа,

2) рабочая область. Позволяет создавать UML диаграммы деятельности,

3) навигатор по диаграмме. Позволяет масштабировать и перемещать по диаграмме деятельности,

4) кнопка «Codegen», запускает процедуру кодогенерации.

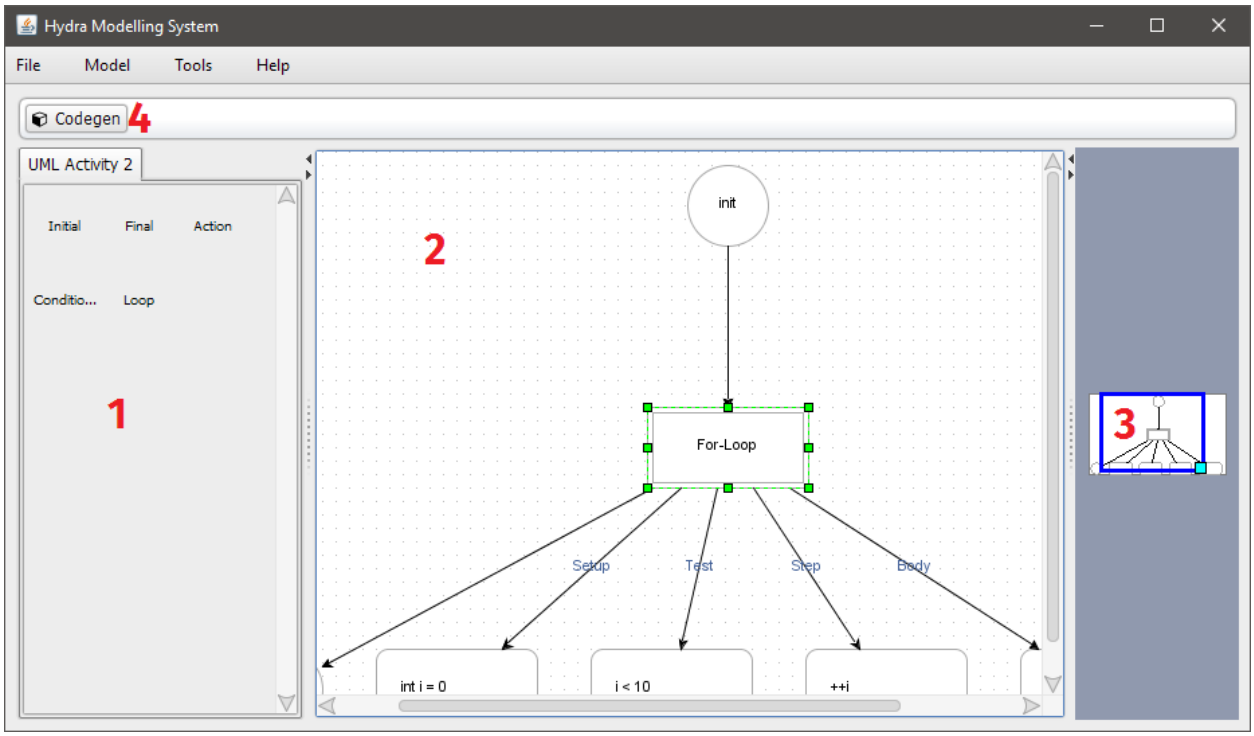

Рисунок 47 – главное окно программы

При двойном клике по узлу «Action» открывается окно редактирования, в котором можно задать значение узла (исходный код) либо сделать его диверсифицируемым. Окно редактирования изображено на рисунках 48 и 49.

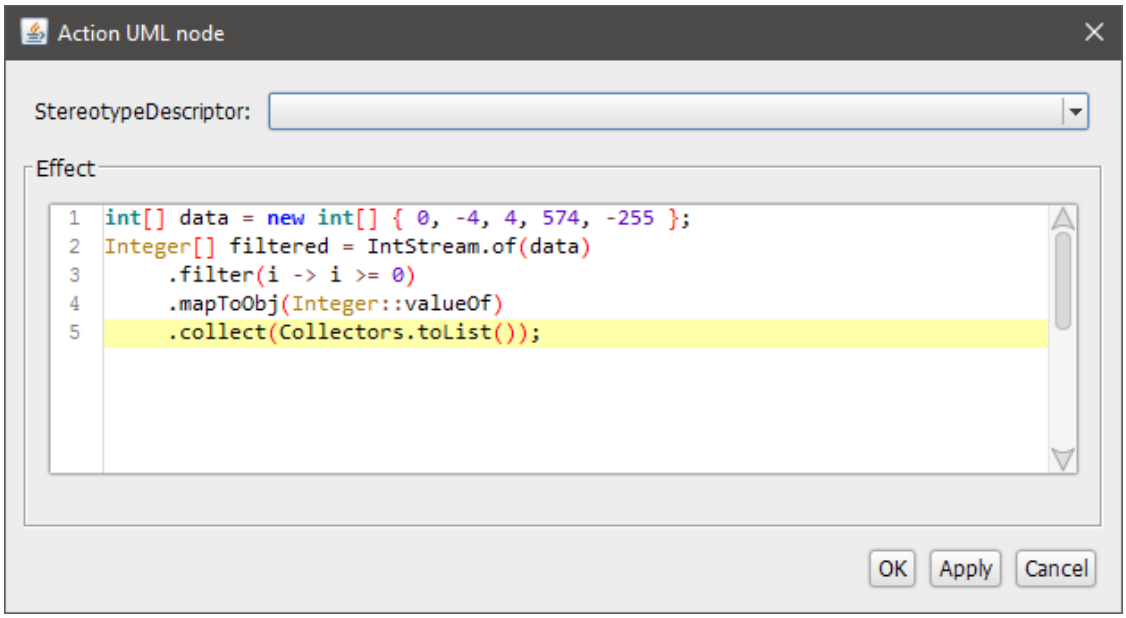

Рисунок 48 – редактор узла Action

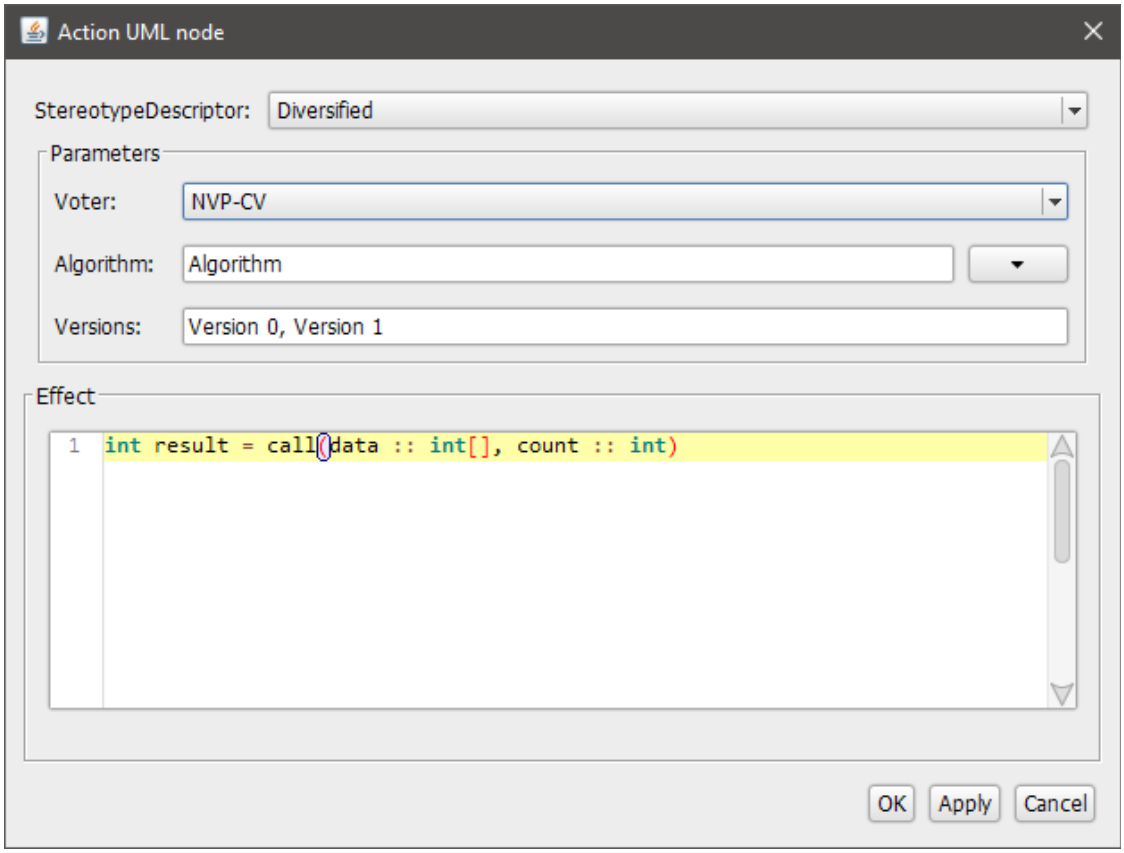

Рисунок 49 – настройка параметров диверсификации

При нажатии на кнопку «Codegen» запускается процедура кодогенерации. По её завершению показывается результат с возможностью сохранить полученный код в файл (см. рисунок 50).

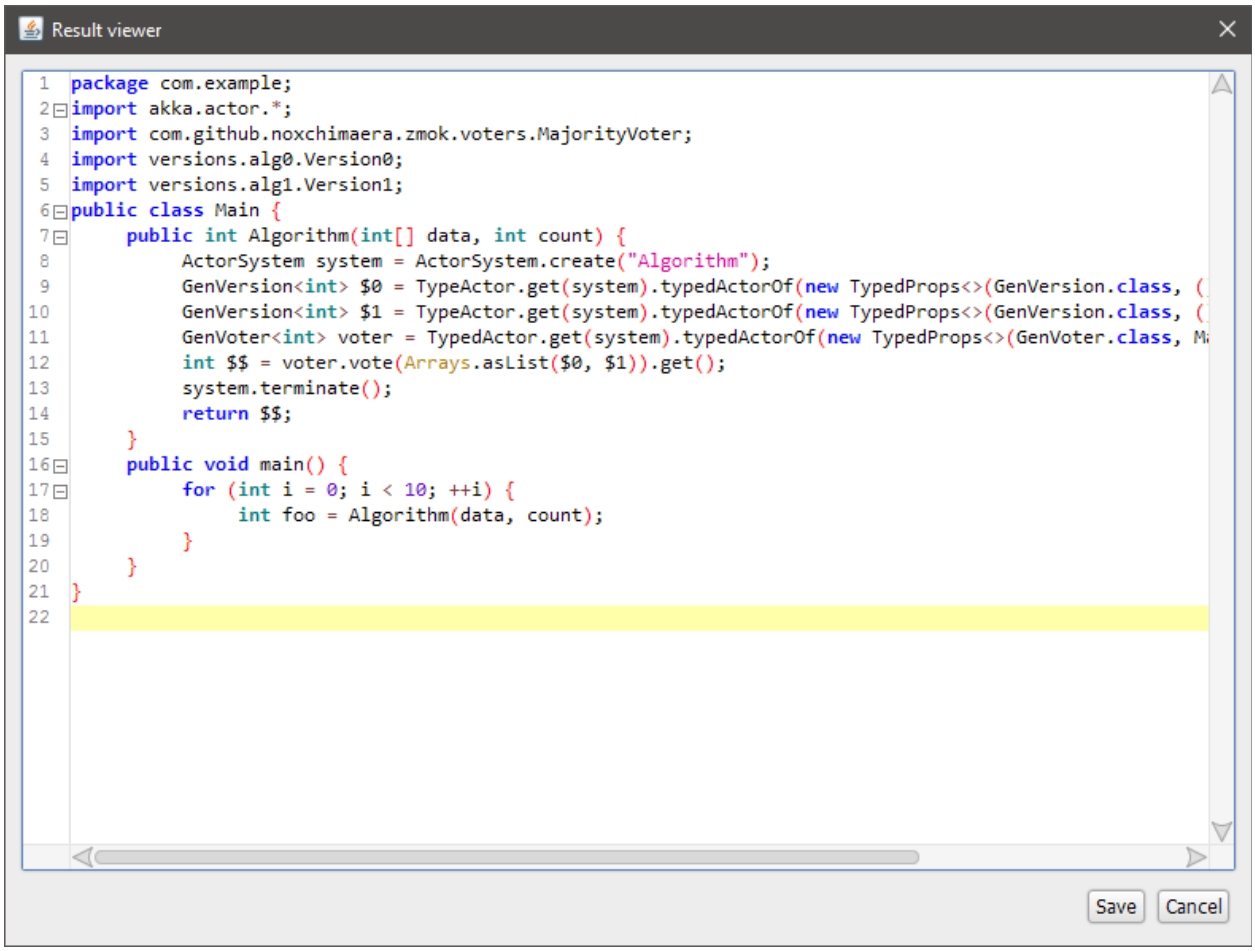

Рисунок 50 – результат кодогенерации

5.5. Вывод

Разработанная система позволяет генерировать диверсифицированное программное обеспечение из диаграммы деятельности с ограничениями.

ЗАКЛЮЧЕНИЕ

В ходе работы были изучены виды и способы диверсификации, изучена методология разработки мультиверсионного программного обеспечения, изучена модель акторов, и предложено применение модели акторов к разработке мультиверсионного программного обеспечения. Была разработана модель кодогенерации из UML диаграмм деятельностей с учётом специфики NVP. Был разработан программный инструментарий для кодогенерации диверсифицированного кода из диаграммы деятельности.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Leveson, N.G. Safeware: System Safety and Computers / N.G. Leveson. — 1-st ed. — Addison-Wesley Professional, 1995. — 704 с.

2. Avižienis, A. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation / A. Avižienis, L. Chen // The Twenty-Fifth International Symposium on Fault-Tolerant Computing. — USA, Pasadena, 1995. — с. 113-119.

3. Грузенкин, Д.В. Модель двухфазной трансляции кода мультиверсий программных модулей / Д.В. Грузенкин, Р.Ю. Царев, А.С. Кузнецов // Фундаментальные исследования. — Пенза : Издательский дом «Академия Естествознания», 2015. — с. 886-890.

4. Compiler-Generated Software Diversity / T. Jackson, B. Salamat, A. Homescu [et al.] // Moving Target Defense. — USA, New York, 2011. — с. 184.

5. Reason, J., The Contribution of Latent Human Failures to the Breakdown of Complex Systems / J. Reason // Philosophical Transactions of the Royal Society of London. – 1990. — т. 327, №1241. — с. 593.

6. Multi-tier diversification in Web-based software application / S. Allier, O. Barais, B. Baudry [et al.] // IEEE Software. — 2015. — т. 32, №11. — с. 83-90.

7. Baudry, B., The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond / B. Baudry, M. Monperrus // ACM Computing Surveys. — USA, New York, 2015. — т. 48, №16. — с. 26,

8. Larsen, P. SoK: Automated Software Diversity / P. Larsen, A. Homescu, S. Brunthaler, M. Franz // IEEE Symposium on Security and Privacy. — 2014. — с. 276-291.

9. Pappas, V., Practical Software Diversification Using In-Place Code Randomization / V. Pappas, M. Polychronakis, A. D. Keromytis // Moving Target Defense II. — USA, New York, 2013. — с. 204.

10. Randell, B. System Structure for Software Fault-Tolerance / B. Randell // IEEE Trans. Soft. Eng. — 1975. — т. SE-1. — с. 220-232.

11. Царёв, Р. Ю., Мультиверсионное программное обеспечение. Алгоритмы голосования и оценка надёжности : монография / Р. Ю. Царёв, А. В. Штарик, Е. Н. Штарик. — Красноярск : Сиб. федер. ун-т, 2013. — 120 с.

12. Bharathi, V., N-Version programming method of Software Fault Tolerance: A Critical Review / V. Bharathi. — Conference of Nonlinear Systems and Dynamics. — India, Kharagpur, 2003. — с. 232.

13. Хоп, Г., Шаблоны интеграции корпоративных приложений / Г. Хор, Б. Вульф. — М.: ООО «И.Д. Вильямс», 2007. — 672 с.

14. Hewitt, C. A universal modular ACTOR formalism for artificial intelligence / C. Hewitt, P. Bishop, R. Steiger // IJCAI'73 Proceedings of the 3rd international joint conference on Artificial Intelligence. — USA, Stanford, 1973. — с. 235-245.

15. Agha, G.A. Actors: A Model of Concurrent Computation In Distributed Systems / G.A. Agha. — 1985. — 190 с.

16. Mitchell, J. C., Concepts in Programming Languages / J. C. Mitchell. — Press Syndicate of the University of Cambridge, 2002. — 584 с.

17. Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Дж. Рамбо, И. Якобсон. — М.: ДМК Пресс, 2006. — 496 с.

18. Рамбо, Дж. UML: специальный справочник / Дж. Рамбо, А. Якобсон, Г. Буч. — СПб.: Питер, 2002. — 656 с.

19. Siebenhaller, M. Drawing activity diagrams / M. Siebenhaller, M. Kaufmann // SoftVis'06 Proceedings of the 2006 ACM symposium on Software visualization. — UK, Brighton, 2006. — с. 159-160.

20. Гома, Х. UML. Проектирование систем реального времени, параллельных и распределённых приложений. — М.: ДМК Пресс, 2011. — 704 с.

21. Ахо, А.В. Компиляторы. Принципы, технологии и инструментарий / А.В. Ахо, М.С. Лам, Р. Сети, Дж.Д. Ульман. — 2-е изд. — М.: Вильямс, 2008. — 1184 с.

22. The Java® Language Specification. Java SE 8 Edition / J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley. — Oracle, 2015. — 768 с.

23. Gupta, M. K. Akka Essentials. — Birmingham: Packt Publishing, 2012. — 334 с.

24. Roestenburg R. Akka in action / R. Roestenburg, R. Bakker, R. Williams. — Manning Publications, 2014. — 394 с.

25. Гамма, Э. Приёмы объектно-ориентированного программирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. — СПб: Питер, 2003. — 368 с.

26. Блох, Дж. Java. Эффективное программирование. — М.: Лори, 2002. — 224 с.