Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана»

Факультет «Робототехника и комплексная автоматизация»
Кафедра «Системы автоматизированного проектирования»

Разработка web-ориентированных CASE инструментариев автоматизации построения исходных кодов графоориентированных решателей

Студент РК6-81Б: *Неклюдов С.А.*

Научный руководитель: *доцент кафедры РК-6, к.ф.-м.н., Соколов А.П.*

Москва, 2019

План выступления

1. Введение
2. Теоретическая часть
3. Программная реализация
4. Тестирование
5. Заключение

Введение

Актуальность

Особенности разработки современных программных систем

- Существенный объем исходного кода
- Высокие трудозатраты на разработку
- Большое количество логических связей между компонентами программной системы

Следствия

- Потребность в оптимизации процессов разработки привела к созданию CASE инструментариев различных типов, в частности – генераторов исходного кода программ
- Генераторы исходного кода активно применяются в крупных системах стратегического и корпоративного характера

Введение

Известные CASE инструментарии

- Системы управления конфигурациями: контроля версий (SVN, GIT, Mercurial …), построения инсталляторов, непрерывной интеграции (CI).
- Средства управления требованиями.
- Средства планирования (например, Redmine, Jira, MS SharePoint и др.).
- Средства анализа программ (анализаторы программ).
- Средства тестирования, отладки, документирования.
- Системы автоматизации построения графических пользовательских интерфейсов.
- **Системы автоматизации генерации кода.**
- И др.

Введение

Цели и задачи работы

Цель: разработать программное обеспечение генерации исходного кода программ на основе графовых моделей алгоритмов.

Задачи:

1. Провести обзор литературы по теме: “Особенности, технологии и методы генерации исходного кода программы на основе графического представления алгоритма”.
2. Разработать архитектуру генератора исходного кода.
3. Реализовать генератор исходного кода.
4. Провести тестирование разработанного ПО.

Теоретическая часть

Графоориентированный подход*

Цели

- Декомпозиция сложных вычислительных задач на более простые подзадачи
- Визуализация алгоритма

Идея

- Представление модели алгоритма в виде ориентированного графа

Модель графа состоит из

- Функций – обработчиков
- Функций предикатов
- Узлов

Рисунок 1. Простейший пример графовой модели алгоритма нахождения нулей функции методом Ньютона

*Соколов А.П., Першин А.Ю. Графоориентированный программный каркас для реализации сложных вычислительных методов. Программирование. – Т.47, №5 – 2019 (в печати).

Теоретическая часть

Модель-ориентированная архитектура

Стадии разработки приложения

- Разработка платформо-независимой модели алгоритма
- Преобразование модели в платформо-зависимую (у M2M генераторов)
- Преобразование модели в исходный код

Модельные преобразования

- M2M – model to model
- M2T – model to text

Рис. 2. Классификация генераторов на основе моделей

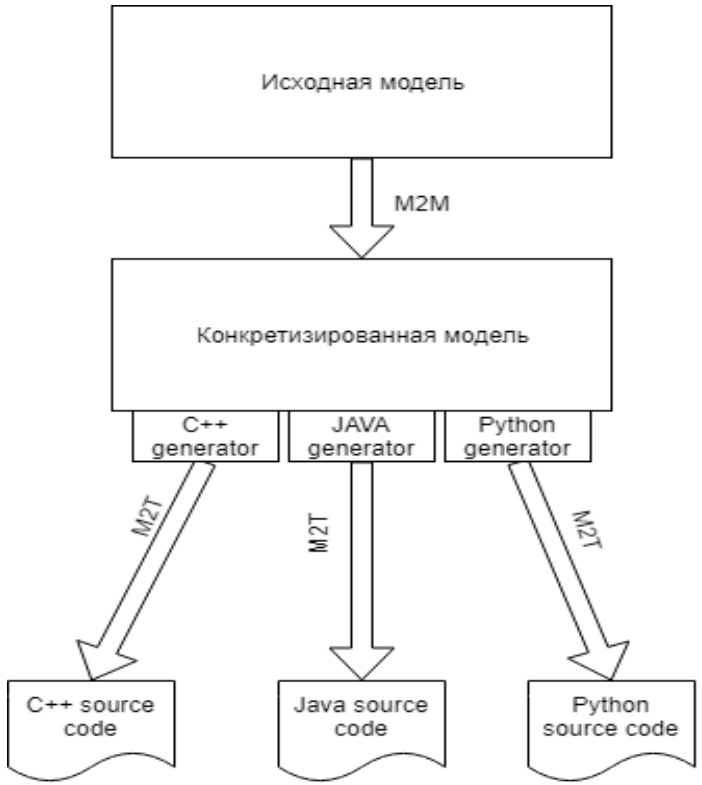

Рисунок 3. M2M генератор

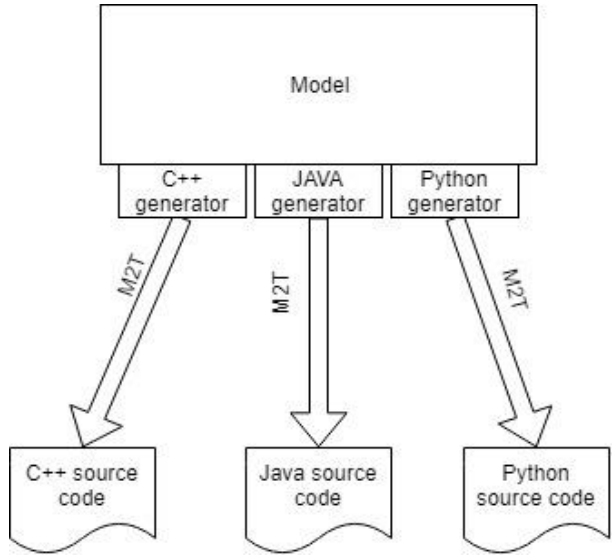

Рисунок 4. M2T генератор

Программная реализация

Используемые технологии

- Генератор исходного кода реализован на языке C++ с использованием вспомогательной библиотеки comsdk*.
- Для построения графовых моделей алгоритмов использовался язык aDOT**.
- Для подготовки исходных данных решателя, исходный код которого будет генерироваться на основе соответствующей графовой модели, предполагается использовать текстовый формат aINI.
- Для запуска процедуры генерации применялось web-приложение comwpc***.

* comsdk – библиотека работы с графовыми моделями.

** aDOT – подмножество языка описания графов DOT.

*** comwpc – клиентское приложение PBC GCD, реализовано на языке Python с использованием Django.

Программная реализация

Схема работы генератора

- Модель представляет собой текстовое описание графового алгоритма в формате обмена данными aDot
- Конфигурационные файлы представляются в формате aIni
- Обход графовой модели реализован в библиотеке comsdk и позволяет находить циклические конструкции.
- Очередь вызовов является структурой типа vector<vector<string>>
- Сериализация – формирование строки типовой конструкции решателя

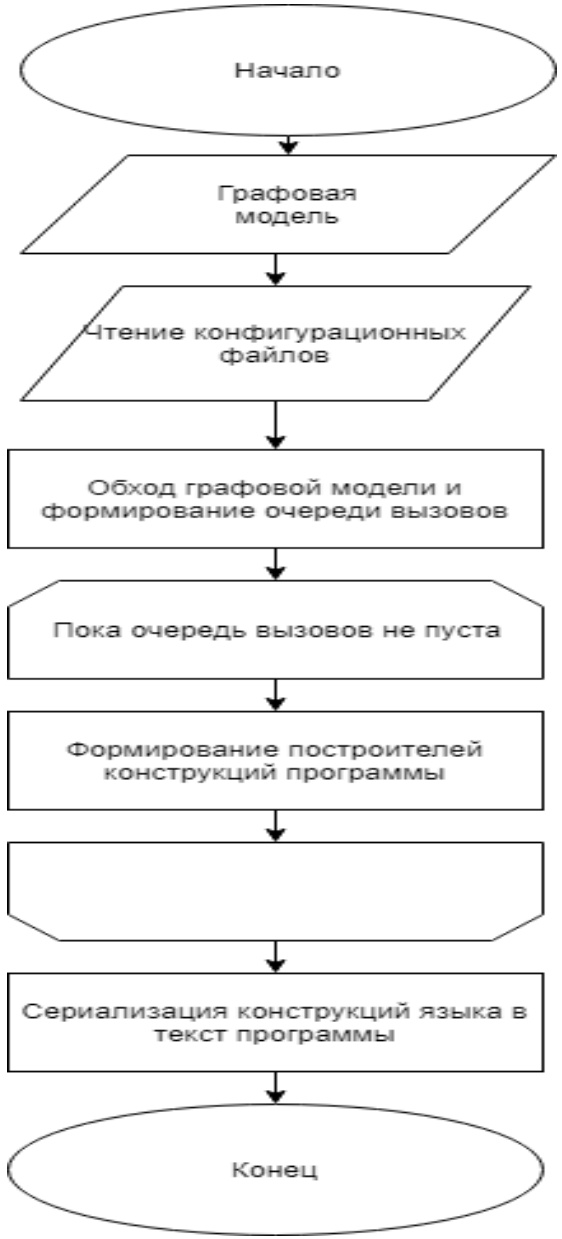

Рисунок 5. Схема алгоритма работы генератора

Программная реализация

Архитектура плагина генерации исходного кода

- programm_generator – ядро программной системы;
- содержит метод serialize() для формирования кода программы;
- класс programm_generator содержит множества объектов – генераторов строк типовых конструкций решателей
- Классы типовых конструкций содержат методы serialize и get_comment()

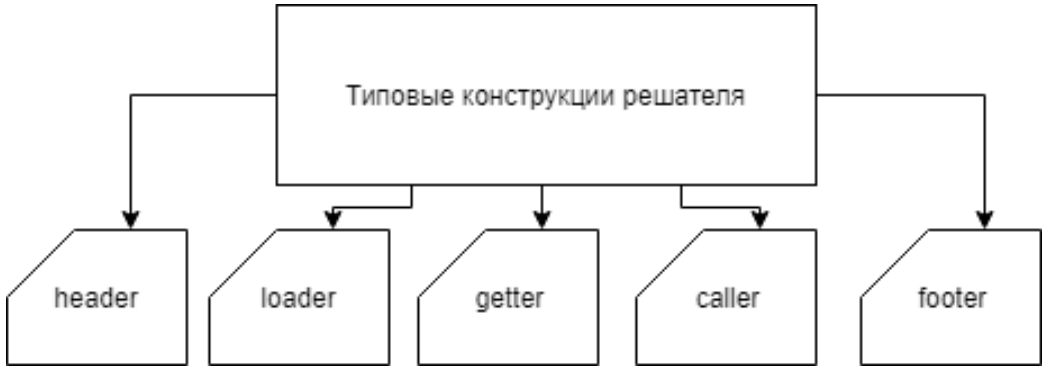

Рисунок 6. Типовые конструкции решателя

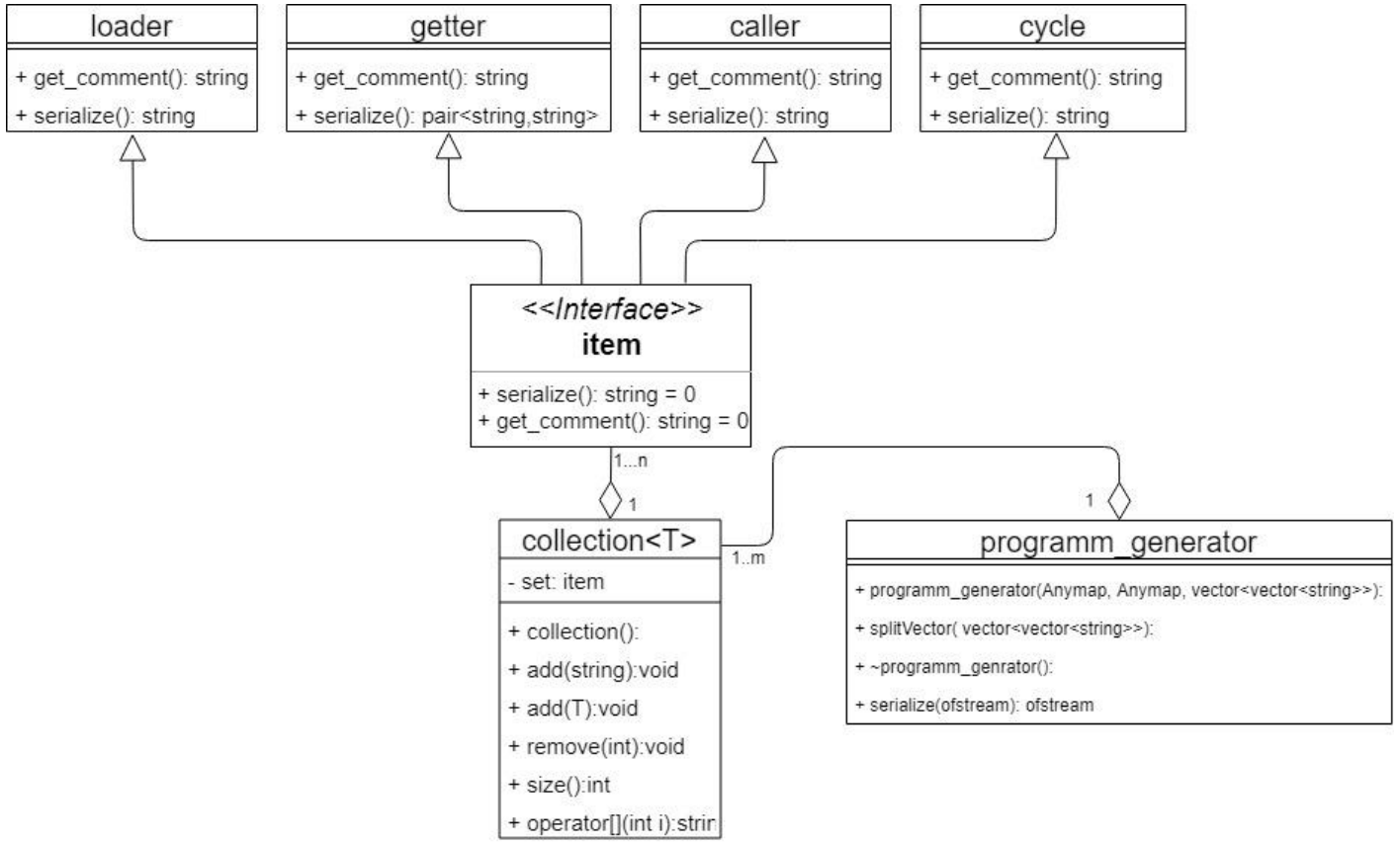

Рисунок 7. Диаграмма классов генератора исходного кода

Программная реализация

Сценарий генерации графический формы ввода - вывода

Сценарий формирования web-страницы на основе файла входных данных aINI

1. Вызов обработчика файла входных данных
2. Чтение файла входных данных
3. Формирование контекста и на основе которого генерируется WEB страница
4. Нажатие клавиши отображения таблицы базы данных (опционально).
5. Запрос таблицы базы данных (опционально)
6. Возврат таблицы и переформирование WEB - страницы (опционально)

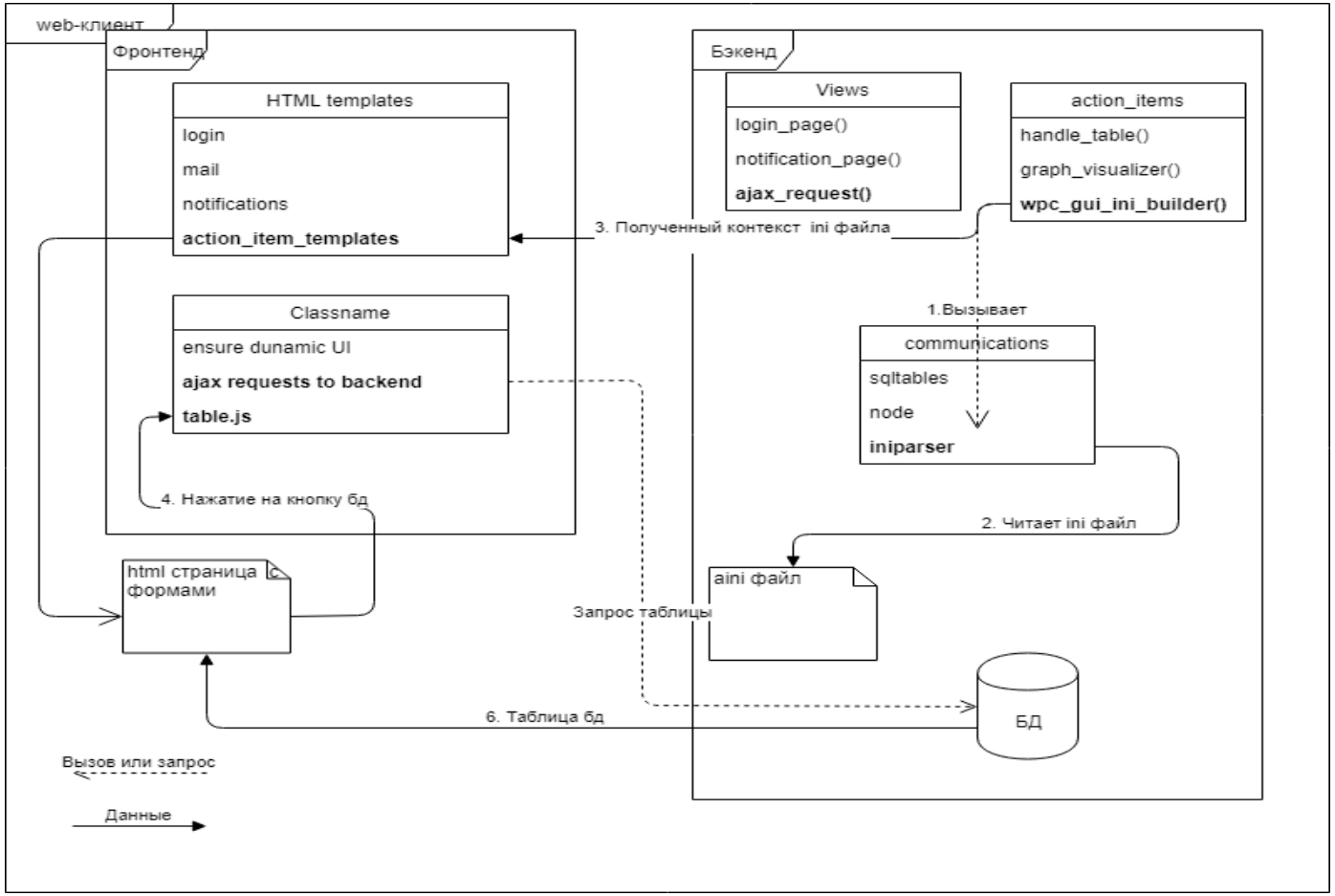

Рисунок 8. Последовательность формирования GUI форм ввода

Тестирование

Интеграция с PBC GCD

Для создания решателя необходимыми являются

- Графовая модель
- Реализованные функции-обработчики и функции-предикаты, используемые в графовой модели
- Кодирование программы решателя

Используя генератор кода разработчик может отказаться от написания решателя

Рисунок 9. Создание решателя без использования генератора

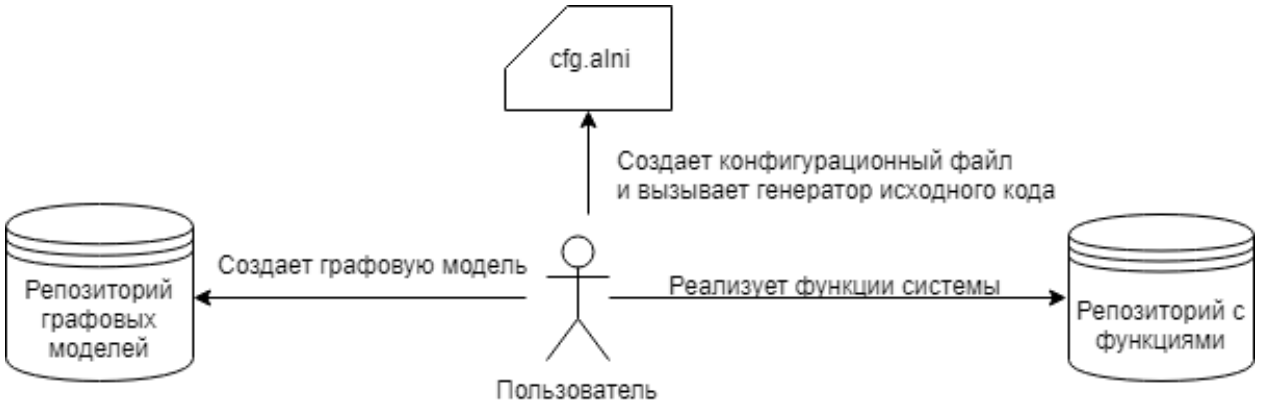

Рисунок 10. Создание решателя с использованием генератора

Тестирование

Модульное тестирование методов реализованных классов

- Реализованные классы-наследники интерфейса **item**: **loader, getter, caller, cycle**
- В каждом классе-наследнике были переопределены методы: а) сериализация объекта класса (**serialize**), б) генерация комментария (**get_comment**).
- Примеры данных (типовые языковые конструкции), хранимых в объектах, указанных классов: **тип переменной, наименование переменной, название библиотек и функций** и пр.

```
Тест 1: Проверка строки загрузки библиотек
//Загрузка библиотеки default_lib
HMODULE lib_default_lib=LoadLibrary(L"default_lib");
Тест 2: Проверка строк поиска функций в библиотеках
//Вызов функции-обработчика default_processor с предикатом default_predicate
auto res = F<proc_default_processor, pred_default_predicate>(default_anymap);if (res!=0) return res;
Тест 3: Проверка строки запуска функции
//Поиск функции обработчика def_proc_name и функции-предиката def_pred_name в библиотеке default_lib
default_processor *proc_def_proc_name = (default_processor *)GetProcAddress(default_lib,"def_proc_name");
default_predicate *pred_def_pred_name = (default_predicate *)GetProcAddress(default_lib, "def_pred_name");
Тест 4: Проверка работы коллекции с loader строками
В коллекцию добавляется строка полученная явным вызовом serialize:
HMODULE lib_default_lib=LoadLibrary(L"default_lib");
В коллекцию добавляется строка полученная loader'ом:
HMODULE lib_default_lib=LoadLibrary(L"default_lib");
Из коллекции удаляется строка:

В коллекцию добавляется строка полученная loader'ом:
//Загрузка библиотеки default_lib HMODULE lib_default_lib=LoadLibrary(L"default_lib");
Тест 5: Цикл
        do {
        a += 1
} while( a < 3 );
```

Рисунок 11. Результаты модульного тестирования*

*Листинги тестовых программ представлены в основном тексте работы

Тестирование

Пример простейшей графовой модели алгоритма

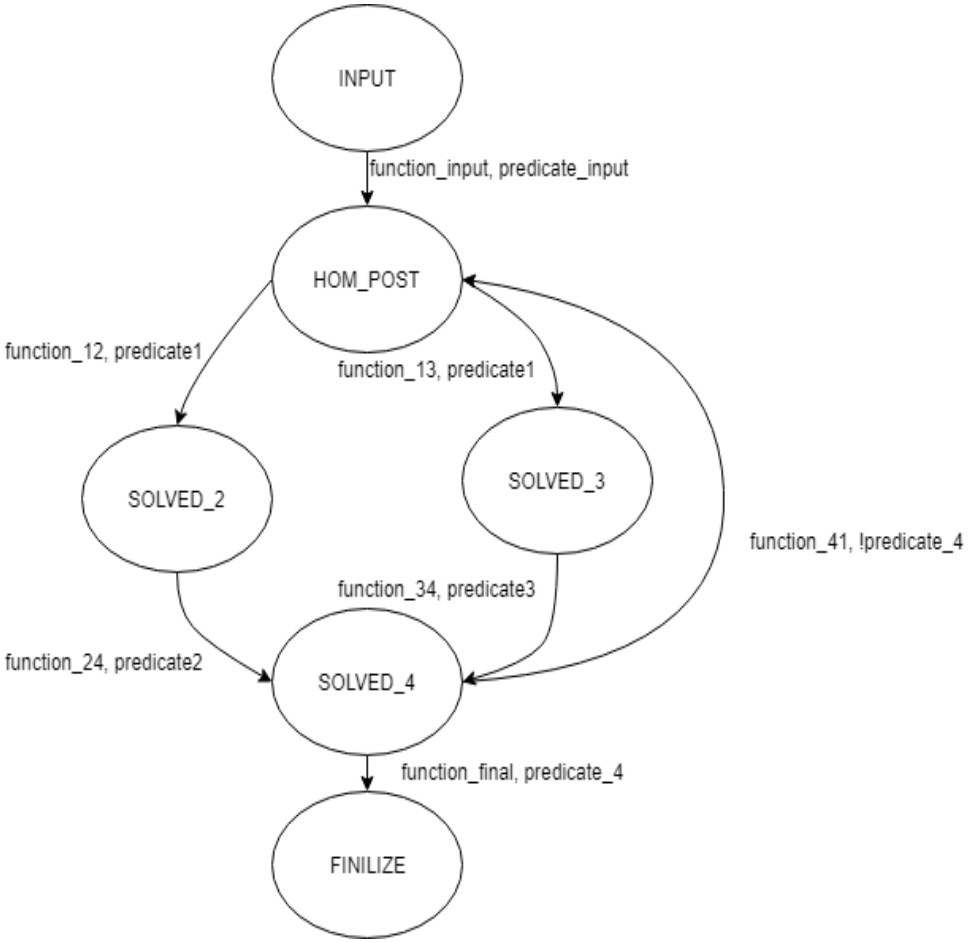

Рисунок 12. Тестовая графовая модель

- Данная графовая **модель содержит цикл**, что приведёт к тому, что результирующая программа должна содержать соответствующие языковые конструкции.
- **Условие выхода из цикла:** проверка значения вычисленной связанной функции-предиката.

Тестирование

Результат генерации для операционной системы семейства MS Windows

Листинг 1. Программа на языке C++, сгенерированная на основе графовой модели

```
#include <anymap.h>
typedef int processorFuncType(AnyMap&);
typedef bool predicateFuncType(const AnyMap&);
template<processorFuncType* tf,predicateFuncType* tp>
int F(AnyMap& p_m)
{
 return (tp(p_m))?tf(p_m):tp(p_m);
}
int main(){
Anymap input(“input.txt”);
Anymap cfg(“cfg.aINi”);
HMODULE lib_name=LoadLibrary(L”name”);
processorFuncType *proc_function_input=(processorFuncType*)GetProcAddress(lib_name,"function_input");
predicateFuncType *pred_predicate_input=(predicateFuncType *)GetProcAddress(lib_name, "predicate_input");
processorFuncType *proc_function_13 = (processorFuncType *)GetProcAddress(lib_name,"function_13");
predicateFuncType *pred_predicate_1 = (predicateFuncType *)GetProcAddress(lib_name, "predicate_1");
processorFuncType *proc_function_24 = (processorFuncType *)GetProcAddress(lib_name,"function_24");
predicateFuncType *pred_predicate_2 = (predicateFuncType *)GetProcAddress(lib_name, "predicate_2");
processorFuncType *proc_function_34 = (processorFuncType *)GetProcAddress(lib_name,"function_34");
predicateFuncType *pred_predicate_3 = (predicateFuncType *)GetProcAddress(lib_name, "predicate_3");
processorFuncType *proc_function_41 = (processorFuncType *)GetProcAddress(lib_name,"function_41");
predicateFuncType *pred_predicate_4 = (predicateFuncType *)GetProcAddress(lib_name, "predicate_4");
processorFuncType *proc_function_final = (processorFuncType *)GetProcAddress(lib_name,"function_final");
auto res = F<proc_function_input, pred_predicate_input>(input);if (res!=0) return res;
do {
auto res = F<proc_function_13, pred_predicate_1>(input);if (res!=0) return res;
auto res = F<proc_function_24, pred_predicate_2>(input);if (res!=0) return res;
auto res = F<proc_function_34, pred_predicate_3>(input);if (res!=0) return res;
auto res = F<proc_function_41, pred_predicate_4>(input);if (res!=0) return res;
} while(!predicate_4);
auto res = F<proc_function_final, pred_predicate_4>(input);if (res!=0) return res;
}
```

Возможности генератора

- Поддержка циклов.
- Условные операторы реализуются за счет применения функций-предикатов.
- Возможности генерации платформа-зависимого исходного кода (поддержка реализована в файле конфигурации).

Заключение

- Был выполнен тщательный анализ источников в результате чего была выявлена потребность в написании программного обеспечения генерации исходного кода
- Разработанная архитектура стала основой для реализации генератора
- Созданное ПО позволит ускорить процесс разработки графоориентированных решателей PBC GCD
- Разработанное ПО послужит ядром подсистемы генерации кода решателей PBC GCD

Спасибо за внимание!