Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ

Заведующий кафедрой _____
(Индекс)

_____ _____
(И.О.Фамилия)
« __ » _____ 20 __ г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине Технология компьютерно-интегрированных производств

Разработка механизма двухпроходной компиляции языка РДО
(Тема курсового проекта)

Студент Александровский К.Д., РК9-92
(Фамилия, инициалы, индекс группы)

График выполнения проекта: 25% к __ нед., 50% к __ нед., 75% к __ нед., 100% к __ нед.

1. Техническое задание

Создание средств разработки двухпроходного компилятора языка РДО и объединение компиляторов языка в единый двухпроходный.

2. Оформление курсового проекта

2.1. Расчетно-пояснительная записка на 33 листах формата А4.

2.2. Перечень графического материала (плакаты, схемы, чертежи и т.п.)

1 лист А1 – Постановка задачи;
2 лист А1 – Сравнительная диаграмма классов компилятора РДО;
3 лист А2 – Диаграмма компонентов «как было»;
4 лист А3 – Диаграмма компонентов «как стало»;
5 лист А3 – Синтаксическая диаграмма компилятора РДО;
6 лист А2 – Алгоритм скрипта генерации грамматик;
7 лист А2 – Алгоритм скрипта запуска bison'а;
8 лист А1 – Результаты.

Дата выдачи задания « ___ » _____ 20__ г.

Руководитель курсового проекта _____ А.В. Урусов
(Подпись, дата) (И.О.Фамилия)

Студент _____ К.Д. Александровский
(Подпись, дата) (И.О.Фамилия)

Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ РК

КАФЕДРА РК-9

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовому проекту на тему:

Разработка механизма двухпроходной компиляции языка РДО

Студент группы РК9-92 _____ К.Д. Александровский
(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ _____ А.В. Урусов
(Подпись, дата) (И.О.Фамилия)

Москва, 2013

Оглавление

1. Перечень сокращений

РП – Рабочий Проект

ТЗ – Техническое Задание

ТП – Технический Проект

ИМ – Имитационное Моделирование, Имитационная Модель

СДС – Сложная Дискретная Система

ЭВМ - Электронная Вычислительная Машина

ОЗУ - Оперативное Запоминающее Устройство

2. Введение

Имитационное моделирование (ИМ) на ЭВМ находит широкое применение при исследовании и управлении сложными дискретными системами (СДС) и процессами, в них протекающими. К таким системам можно отнести экономические и производственные объекты, морские порты, аэропорты, комплексы перекачки нефти и газа, ирригационные системы, программное обеспечение сложных систем управления, вычислительные сети и многие другие.

Интеллектуальное имитационное моделирование, характеризующиеся возможностью использования методов искусственного интеллекта и, прежде всего, знаний при принятии решений в процессе имитации, при управлении имитационным экспериментом, при реализации интерфейса пользователя, создании информационных банков ИМ, использовании нечетких данных, снимает часть проблем использования ИМ.

ИМ является эффективным, но и не лишенным недостатков, методом. Трудности использования ИМ, связаны с обеспечением адекватности описания системы, интерпретацией результатов, обеспечением стохастической сходимости процесса моделирования, решением проблемы размерности и т.п. К проблемам применения ИМ следует отнести также и большую трудоемкость данного метода.

Разработка интеллектуальной среды имитационного моделирования РДО – «Ресурсы, Действия, Операции», выполнена в Московском Государственном Техническом Университете им. Н.Э. Баумана на кафедре "Компьютерные системы автоматизации производства". Причинами создания РДО явились требования к универсальности ИМ относительно классов моделируемых систем и процессов, легкости модификации

моделей, а также моделирования сложных систем управления совместно с управляемым объектом (включая использование ИМ в управлении в реальном масштабе времени). Таким образом, среда РДО стала решением указанных выше проблем ИМ и обеспечила исследователя и проектировщика новыми возможностями.

Программный комплекс RAO-studio предназначен для разработки и отладки имитационных моделей на языке РДО. Основные цели данного комплекса - обеспечение пользователя легким в обращении, но достаточно мощным средством разработки текстов моделей на языке РДО, обладающим большинством функций по работе с текстами программ, характерных для сред программирования, а также средствами проведения и обработки результатов имитационных экспериментов.

3. Предпроектное исследование

3.1. Основные положения языка РДО

В основе системы РДО лежат следующие положения:

- Все элементы сложной дискретной системы (СДС) представлены как ресурсы, описываемые некоторыми параметрами.
- Состояние ресурса определяется вектором значений всех его параметров; состояние СДС – значением всех параметров всех ресурсов.
- Процесс, протекающий в СДС, описывается как последовательность целенаправленных действий и нерегулярных событий, изменяющих определенным образом состояния ресурсов; действия ограничены во времени двумя событиями: событиями начала и конца.
- Нерегулярные события описывают изменение состояния СДС, непредсказуемые в рамках продукционной модели системы (влияние внешних по отношению к СДС факторов либо факторов, внутренних по отношению к ресурсам СДС). Моменты наступления нерегулярных событий случайны.
- Действия описываются операциями, которые представляют собой модифицированные продукционные правила, учитывающие временные связи. Операция описывает предусловия, которым должно удовлетворять состояние участвующих в операции ресурсов, и правила изменения ресурсов в начале и конце соответствующего действия.

При выполнении работ, связанных с созданием и использованием ИМ в среде РДО, пользователь оперирует следующими основными понятиями:

Модель – совокупность объектов РДО-языка, описывающих какой-то реальный объект, собираемые в процессе имитации показатели, кадры анимации и графические элементы, используемые при анимации, результаты трассировки.

Прогон – это единая неделимая точка имитационного эксперимента. Он характеризуется совокупностью объектов, представляющих собой исходные данные и результаты, полученные при запуске имитатора с этими исходными данными.

Проект – один или более прогонов, объединенных какой-либо общей целью. Например, это может быть совокупность прогонов, которые направлены на исследование одного конкретного объекта или выполнение одного контракта на имитационные исследования по одному или нескольким объектам.

Объект – совокупность информации, предназначенной для определенных целей и имеющая смысл для имитационной программы. Состав объектов обусловлен РДО-методом, определяющим парадигму представления СДС на языке РДО.

Объектами исходных данных являются:

- типы ресурсов (с расширением .rtp);
- ресурсы (с расширением .rss);
- события (с расширением .evn);
- образцы активностей (с расширением .pat);
- точки принятия решений и процессы обслуживания (с расширением .dpt);
- константы, функции и последовательности (с расширением .fun);
- кадры анимации (с расширением .frm);
- требуемая статистика (с расширением .pmd);
- прогон (с расширением .smr);
- процессы обслуживания (с расширением .prc).

Объекты, создаваемые РДО-имитатором при выполнении прогона:

- результаты (с расширением .pmv);
- трассировка (с расширением .trc). [2]

3.2. Понятие компилятора

Компилятор — это программа, которая считывает текст программы, написанной на одном языке — исходном, и транслирует (переводит) его в эквивалентный текст на другом языке — целевом. [4, с. 29]

Компилятор отображает исходную программу в семантически эквивалентную ей целевую программу. Это отображение разделяется на две части: анализ и синтез. *Анализ* разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем он использует эту структуру для создания промежуточного представления исходной программы. Если анализ обнаруживает, что исходная программа неверно составлена синтаксически либо дефектна семантически, он должен выдать информативные сообщения об этом, чтобы пользователь мог исправить обнаруженные ошибки. Анализ также собирает информацию об исходной программе и сохраняет ее в структуре данных, именуемой таблицей символов, которая передается вместе с промежуточным представлением синтезу. *Синтез* строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов. Анализ часто называют начальной стадией (front end), а синтез — заключительной (back end). [4, с. 33]

3.3. Двухпроходная компиляция

Процесс компиляции представляет собой последовательность фаз, каждая из которых преобразует одно из представлений исходной программы в другое. Фазы связаны с логической организацией компилятора. При реализации работа разных фаз может быть сгруппирована в проходы (pass), которые считывают входной файл.

Двухпроходный (многопроходный) компилятор обрабатывает исходный код несколько раз, а каждый следующий этап компиляции

учитывает работу предыдущего, улучшая, таким образом, проработку исходного кода программы. [5]

Преимуществом многопроходных компиляторов является возможность написания более гибких конструкций (к примеру, использование названий переменных до их объявления при отсутствии необходимости предекларации).

3.4. Компиляторы языка РДО

При разработке среды РДО используется генератор синтаксических LALR-анализаторов GNU bison (*LALR – восходящий алгоритм синтаксического разбора*) в связке с лексическим анализатором GNU lexer.

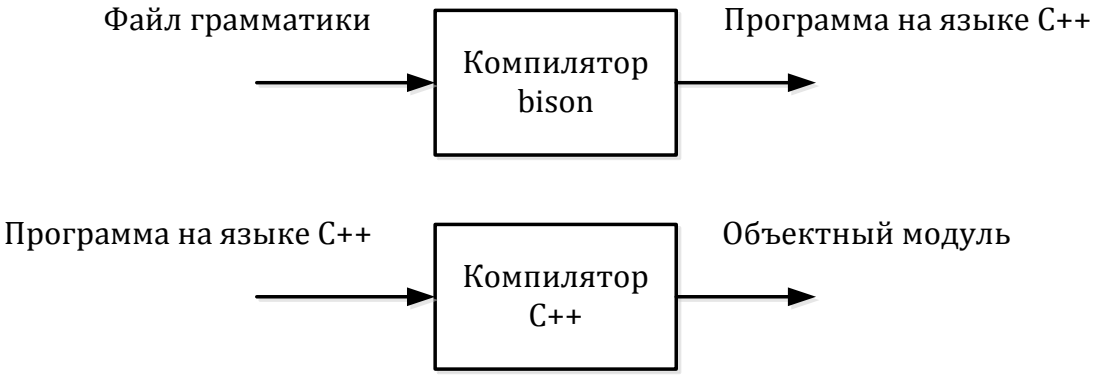

Рис. 1. Создание компилятора с помощью bison [4, стр. 365]

Компиляторы языка РДО описаны в файлах грамматик bison, имеющих расширение ".y" и содержащих описание языка через последовательности терминальных и нетерминальных символов (Рис. 1).

> *«Дублирование – главный враг хорошо спроектированной системы. Его последствия – лишняя работа, лишний риск и лишняя избыточная сложность»* [6, с. 203].

В силу того, что среда РДО реализует несколько подходов к моделированию, а язык РДО в течение длительного времени подвергался влиянию различных парадигм (при этом обеспечивая обратную

совместимость с его ранними версиями), компиляция исходного кода моделей основана на многочисленных независимых компиляторах, каждый из которых обрабатывает определенные синтаксические конструкции языка.

Изначально эти конструкции также были независимыми друг от друга, однако с введением таких возможностей, как процедурный язык программирования, в файлах грамматик генератора синтаксических анализаторов bison стали появляться большие объемы одинакового кода, замедляющие процесс разработки.

При этом некоторые компиляторы языка РДО уже сделаны в многопроходном варианте, включая в себя этапы позднего связывания и этапы предкомпиляции, однако распределение функционала, возложенного на каждый этап, пока не является оптимальным с точки зрения устоявшихся методик проектирования компиляторов.

Также зависимость от большого количества разных компиляторов мешает развитию системы и языка в целом, выступая блокирующим фактором в реализации новых концепций имитационного моделирования.

3.5. Постановка задачи

Таким образом, наиболее удачным вариантом перепроектирования процесса компиляции в РДО станет реализация двухпроходного компилятора, единого для всех синтаксических конструкций, присущих языку.

Инфографика по данным изменениям в системе представлена на листе А1 курсового проекта «Постановка задачи».

4. Концептуальный этап проектирования системы

4.1. Диаграмма компонентов

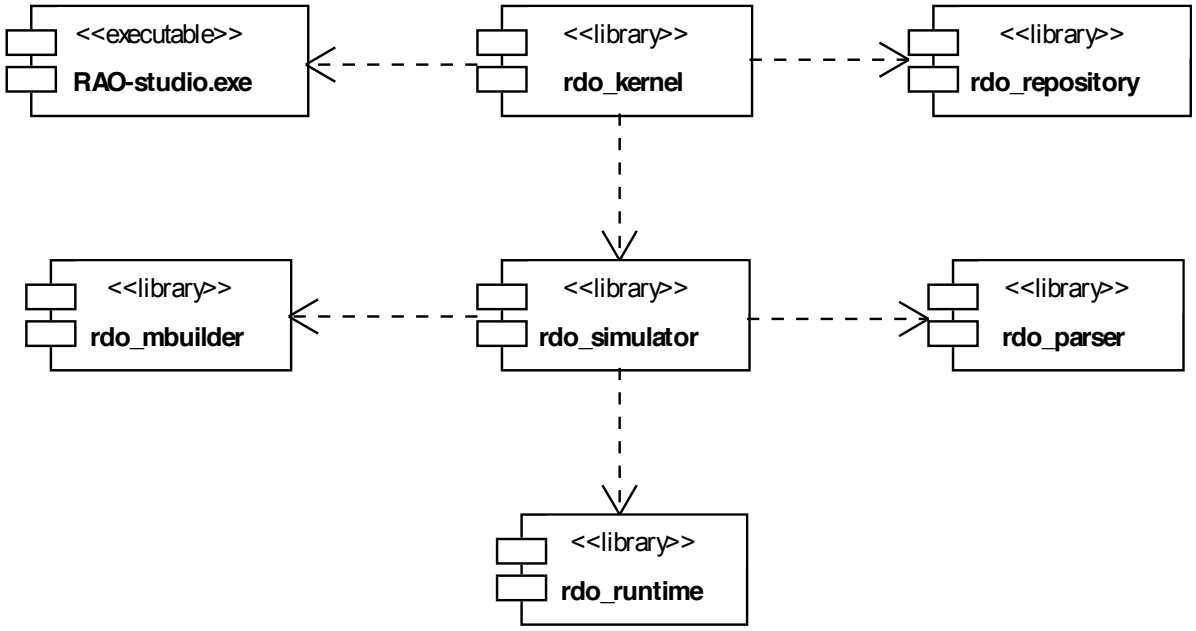

Рис. 2. Упрощенная диаграмма компонентов

Функционал представленных на диаграмме компонентов (Рис. 2):

rdo_kernel реализует функции ядра системы. Не изменяется при разработке системы.

RAO-studio.exe реализует графический интерфейс пользователя. Не изменяется при разработке системы.

rdo_repository реализует управление потоками данных внутри системы и отвечает за хранение и получение информации о модели. Не изменяется при разработке системы.

rdo_mbuilder реализует функционал, используемый для программного управления типами ресурсов и ресурсами модели. Не изменяется при разработке системы.

rdo_simulator управляет процессом моделирования на всех его этапах. Он осуществляет координацию и управление компонентами rdo_runtime и rdo_parser. Не изменяется при разработке системы.

rdo_parser производит лексический и синтаксический разбор исходных текстов модели, написанной на языке РДО. Модернизируется при разработке системы.

rdo_runtime отвечает за непосредственное выполнение модели, управление базой данных и базой знаний. Не изменяется при разработке системы.

Объекты компонента **rdo_runtime** инициализируются при разборе исходного текста модели компонентом **rdo_parser**.

В данной работе все модификации необходимо произвести лишь над компонентом **rdo_parser**, так как в нем целиком определяется процесс компиляции.

4.2. Реализация механизма двухпроходной компиляции

Для общего понимания принципов работы компилятора среды РДО рассмотрим пример принципиальной структуры файла грамматики генератора компиляторов bison в части правил трансляции:

```
line   : expr '\n' { printf("%d\n", $1); }
expr   : expr '+' term { $$ = $1 + $3; }
       | term
       ;
Term   : term '*' factor { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')' { $$ = $2; }
       | DIGIT
       ;
```

В продукциях bison строки букв и цифр без кавычек, не объявленные как токены, считаются нетерминалами. Отдельный символ в одинарных кавычках, например “c”, представляет терминальный символ c .

Альтернативные тела продукций разделяются вертикальной чертой, а за каждой продукцией с ее альтернативами и семантическими действиями ставится точка с запятой.

Семантические действия bison (инструкции в фигурных скобках) представляют собой последовательности инструкций на языке C++, выполняемые для каждой продукции [4, стр. 367].

Однако недостатком bison'а является однопроходность генерируемых им компиляторов, поэтому реализация многопроходного компилятора требует написания отдельного файла грамматик для каждого прохода, причем строение этих файлов в части правил трансляции будет совершенно одинаковым, а различаться будут лишь семантические действия, содержащие код на языке C++.

Подобный обходной прием также является дублированием кода, что является анти-паттерном проектирования, и влечет за собой проблемы, описанные в **3.4.**

В данном случае приемлемым решением этой проблемы будет генерация исходного кода на более высоком уровне, то есть создание файлов грамматик для каждого прохода компиляции из общего исходника. В таком случае в блоки семантических действий необходимо внести новые конструкции, которые позволят разделить код, предназначенный для первого или второго прохода компилятора, таким образом, что на вход bison'а будут поступать два файла, содержащие код, предназначенный лишь для определенного прохода компиляции.

Это позволит получить гибкий механизм создания двухпроходного компилятора без необходимости дублирования кода и редактирования сразу нескольких файлов грамматик при внесении изменений в язык РДО.

4.3. Объединение компиляторов РДО

Для создания единого компилятора языка РДО потребуется разработка обобщенной синтаксической структуры всех синтаксических конструкций

языка и объединение всех файлов грамматик на базе механизма двухпроходной компиляции.

Таким образом, будет решен второй аспект проблемы дублирования кода, а в среде РДО станет доступным последовательное написание в моделях синтаксических конструкций из разных объектов исходных данных языка РДО. Такой подход при сохранении старой парадигмы разделения модели на различные типы сущностей (являющейся наследием первых версий языка) даст возможность написания более гибких, логичных и читаемых моделей, а также позволит ввести понятие исходного кода на языке РДО и превратить среду РДО в интерпретатор имитационных моделей.

Диаграммы компонентов до и после внесения данных изменений (объединение файлов грамматик) изображены на листах А2 «Диаграмма компонентов “как было”» и А3 «Диаграмма компонентов “как стало”» курсового проекта.

5. Формирование ТЗ

5.1. Основания для разработки

Задание на курсовой проект.

5.2. Общие сведения.

В среде РДО разрабатывается и вводится механизм двухпроходной компиляции языка.

5.3. Назначение и цели развития системы

Основная цель работы – создание средств разработки двухпроходного компилятора языка РДО и объединение компиляторов языка в единый двухпроходный.

5.4. Характеристики объекта автоматизации

РДО – язык имитационного моделирования, включающий все три основных подхода описания дискретных систем: процессный, событийный и сканирования активностей.

5.5. Требования к системе

5.5.1. Требования к функциональным характеристикам

- Возможность написания обобщенного файла грамматик, реализующего механизм двухпроходной компиляции;
- Предоставление средств для отладки исходного кода на языке C++, содержащегося в обобщенном файле грамматик;
- Объединение основных компиляторов синтаксиса РДО в единый двухпроходный компилятор при неизменности результатов моделирования в уже написанных имитационных моделях.

5.5.2. Требования к надежности

Основное требование к надежности направлено на поддержание в исправном и работоспособном состоянии ЭВМ, на которой происходит использование программного комплекса RAO-studio.

5.5.3. Условия эксплуатации

Аппаратные средства должны эксплуатироваться в помещениях с выделенной розеточной электросетью 220В±10%, 50 Гц с защитным заземлением.

5.5.4. Требования к составу и параметрам технических средств

Программный продукт должен работать на компьютерах со следующими характеристиками:

- объем ОЗУ не менее 1 Гб;
- объем жесткого диска не менее 20 Гб;
- микропроцессор с тактовой частотой не менее 1 ГГц;
- монитор с разрешением от 800*600 и выше;

5.5.5. Требования к информационной и программной совместимости

Данная система должна работать под управлением операционных систем Windows XP, Windows 7, Windows 8, а также Ubuntu Linux (или ему подобных).

5.5.6. Требования к маркировке и упаковке

Не предъявляются.

5.5.7. Требования к транспортированию и хранению

Не предъявляются.

5.5.8. Порядок контроля и приемки

Контроль работоспособности программного комплекса осуществляется на базе системы автоматического тестирования разрабатываемого программного обеспечения Jenkins.

6. Технический этап проектирования системы

6.1. Разработка механизма двухпроходной компиляции

Для отделения синтаксических конструкций, связанных с каждым проходом компиляции, в разрабатываемом файле генерации грамматик был выбран следующий синтаксис – блок, соответствующий определенному проходу, выделяется фигурными скобками (“{” и “}”) и предшествующим им ключевым словом “#PASS1” или “#PASS2” для первого и второго прохода компиляции соответственно. Пример фрагмента правил трансляции в обобщенном файле грамматик будет выглядеть следующим образом:

```
pat_header
: RDO_Pattern RDO_IDENTIF_COLON RDO_event pat_trace
{
      LPRDOValue pName = PARSER->stack().pop<RDOValue>($2);
      ASSERT(pName);

      #PASS1
      {
            LPRDOPATPattern pPattern =
               rdo::Factory<RDOPatternOperation>::create(
                  pName->src_info(), $4 != 0);
            ASSERT(pPattern);
      }

      #PASS2
      {
            LPRDOPATPattern pPattern = PARSER->findPATPattern(
               pName->value().getIdentificator());
            ASSERT(pPattern);
            pPattern->pushContext();
      }

      $$ = PARSER->stack().push(pPattern);
}
```

Таким образом, для каждого из проходов компиляции в данном примере выполнится лишь свой код, заключенный в фигурные скобки, а также общие команды в начале и в конце семантического действия.

Для генерации двух файлов грамматик из одного обобщенного файла в среде разработки потребуется совершение двух последовательных действий:

- Генерация двух файлов грамматик, соответствующих каждому из проходов компиляции
- Запуск генератора синтаксических анализаторов bison для каждого из сгенерированных файлов для создания объектных модулей языка C++ (см. **3.4**).

6.2. Объединение компиляторов РДО

6.2.1. Изменения в архитектуре компонента rdo_parser

Для обеспечения работоспособности предложенного решения по единому двухпроходному компилятору языка РДО в компонент **rdo_parser** требуется внесение различных изменений, среди которых:

- Изменение механизма запуска компиляторов в классе `RDOParser` (класс, отвечающий за процесс компиляции в РДО)
- Удаление больше не требующихся в силу новой концепции классов, связанных с классом `RDOParser`
- Перенос post-компиляторов из списка компиляторов в отдельные методы класса `RDOParser`, вызываемые лишь перед самым запуском имитационной модели, а так же удаление соответствующих им классов

Все изменения в архитектуре классов, отвечающих за процесс компиляции, показаны на диаграмме классов, выполненной листе А1 курсового проекта «Сравнительная диаграмма классов компилятора РДО». На данной диаграмме, для наглядности и облегчения восприятия, вносимые изменения отмечены цветом. Красным выделены сущности, подвергаемые удалению, а зеленым – разрабатываемые в классе `RDOParser` новые методы.

6.2.2. Синтаксис корневого компилятора РДО

Для возможности написания любых объектов исходных данных РДО в исходном коде модели необходимо при объединении создать в правилах трансляции файла грамматики терминальный символ корневого компилятора, раскрывающийся до каждого из объектов в РДО (объявление ресурсов, их типов, событий, точек принятия решений и т.д.).

Таким образом, используя праворекурсивную подстановку, получим следующее правило трансляции:

```
rdo_compiler
      : /* empty */
      | rtp_main rdo_compiler
      | rss_main rdo_compiler
      | pat_main rdo_compiler
      | dpt_main rdo_compiler
      | prc_main rdo_compiler
      | frm_main rdo_compiler
      | fun_list rdo_compiler
      | pmd_main rdo_compiler
      | error
      {
            PARSER->error().error(RDOParserSrcInfo(),
                                  "Синтаксическая ошибка");
      }
      ;
```

Синтаксическая диаграмма, соответствующая корневому терминальному символу в обобщенном файле грамматики, представлена на листе А3 «Синтаксическая диаграмма компилятора РДО» курсового проекта.

7. Рабочий этап проектирования системы

7.1. Реализация механизма двухпроходной компиляции

Для решения задач генерации файлов грамматик на основе обобщенного и запуска генератора компиляторов bison для каждого из них было решено использовать скриптовый (интерпретируемый) высокоуровневый язык программирования общего назначения Python, как наиболее подходящий в силу своих преимуществ (скорость разработки, гибкий расширяемый за счет модулей функционал и др.).

7.1.1. Разработка скрипта генерации файлов грамматик

Скрипт генерации файлов грамматик был написан для обеспечения следующих возможностей:

- Отделение блоков, соответствующих каждому из проходов компиляции и запись их в соответствующий файл при подавлении блоков с кодом, не относящимся к данному проходу
- В сгенерированном файле для обеспечения корректного перенаправления сообщений отладки и компиляции языка C++ сохраняется положение строк относительно обобщенного файла грамматик (вырезанные блоки из не относящегося к данному прохода заменяются эквивалентным количеством пустых строк)

На вход скрипт получает следующие параметры:

- Путь к входному файлу (расширение “.yx”, без ключа)
- Имя сгенерированного файла первого прохода (с ключом “-y1”)
- Имя сгенерированного файла второго прохода (с ключом “-y2”)

Алгоритм скрипта представлен на листе А2 «Алгоритм скрипта генерации грамматик» курсового проекта.

7.1.2. Разработка скрипта запуска генератора компиляторов bison

Скрипт запуска генератора компиляторов bison выполняется средой разработки сразу после выполнения скрипта генерации, и в качестве входных параметров получает:

- Параметры, аналогичные таковым в скрипте генерации (имена обобщенного файла грамматик и имена генерируемых)
- Путь к исполняемому файлу GNU bison
- Имена файлов с исходным кодом программ на языке C++, генерируемых bison'ом, а также имена для подстановки в переменную имени этих файлов
- Опциональный флаг создания файла определений bison

В процессе работы скрипт:

- Формирует команды запуска bison’а (генерации кода программ на языке C++)
- Собирает поток вывода запускаемых процессов
- Путем сравнения производит валидацию потока вывода bison’а для каждого из сгенерированных предыдущим скриптом файлов грамматик
- Для распознавания ошибок и предупреждений в среде разработки в полученном выводе заменяет пути к сгенерированным файлам грамматик на путь к обобщенному файлу грамматик
- Производит аналогичную операцию в сгенерированных программах на языке C++ (пути директивы компилятора “`#line`”, позволяющей производить пошаговую отладку в исходном обобщенном файле грамматик и ускоряющей процесс отладки разрабатываемого ПО)

Алгоритм работы этого скрипта представлен на листе А2 «Алгоритм скрипта запуска bison’а» курсового проекта.

7.1.3. Интеграция в систему автоматизации сборки ПО

В разработке РДО используется система автоматизации сборки программного обеспечения CMake, позволяющая автоматически выполнять все этапы сборки программного обеспечения на разных платформах. Для интеграции разработанных скриптов генерации грамматик и запуска bison'а был написан модуль `split-bison.cmake`, позволяющий запустить эти скрипты с необходимыми параметрами в качестве одного из шагов сборки среды РДО. Его листинг представлен в **Приложении В**.

7.2. Объединение компиляторов языка РДО

7.2.1. Объединение файлов грамматик

В процессе объединения файлов грамматик (путем слияния каждого следующего файла с обобщенным двухпроходным файлом грамматик ".yx") были решены такие задачи, как устранение избыточной рекурсии в правилах трансляции нижних по отношению к корневому правилу уровней и разрешение конфликтов «свертка/свертка» и «перенос/свертка», вызванных неоднозначностью полученной грамматики из-за неявного дублирования правил. (См. [4, стр. 368]).

Объем итогового файла получился почти в два раза меньше по объему относительно суммы исходных файлов грамматик, что свидетельствует о том, что цель устранения избыточности кода была достигнута, таким образом, код стал чище, а процесс проектирования системы несколько проще.

Также благодаря единому компилятору стало возможным последовательное описание различных сущностей в одной из вкладок среды РДО, что в будущем даст возможность отказаться от старой системы, где исходный код модели жестко привязан к каждой вкладке, в пользу более гибкой системы, в которой среда РДО будет выступать своего

рода интерпретатором имитационных моделей, оформленных в виде файла с исходным кодом на языке РДО или же группы таких файлов.

Пример запуска такой модели представлен оконной формой на листе A1 «Результаты» курсового проекта.

7.2.2. Изменения в компоненте rdo_parser

В ходе реализации архитектурных изменений в классе RDOParser появился список компиляторов Compilers, хранящий объекты компиляции.

За ненадобностью в компоненте rdo_parser были удалены следующие классы:

- `RDOParserContainer`
- `RDOParserContainerSMRInfo`
- `RDOParserContainerModel`
- `RDOParserContainerCorba`
- `RDOParserTemplate`
- `RDOParserRSS`

А также классы, связанные с компиляторами позднего связывания (т.н. пост-компиляторами):

- `RDOParserSMRPost`
- `RDOParserRSSPost`
- `RDOParserRTPPost`
- `RDOParserEVNPost`

Код, выполняемый при компиляции модели на этих этапах, был перенесен из отдельных компиляторов в методы `runRSSPost()`, `runSMRPost()` и `runRTPPost()`, а также общий метод `beforeRun()`, их вызывающий. Код компилятора `RDOParserEVNPost` был убран в силу его распределения по проходам компиляции.

Листинг Post-методов, запускаемых перед началом моделирования:

```
void RDOParser::runRSSPost()
{
	//! В режиме совместимости со старым РДО создаем ресурсы по
		номерам их типов, а не по номерам самих ресурсов из RSS
#ifdef RDOSIM_COMPATIBLE
	STL_FOR_ALL_CONST(getRTPResTypes(), rtp_it)
	{
#endif
		STL_FOR_ALL_CONST(getRSSResources(), rss_it)
		{
#ifdef RDOSIM_COMPATIBLE
			if ((*rss_it)->getType() == *rtp_it)
			{
#endif
				rdo::runtime::LPRDOCalc calc =
					(*rss_it)->createCalc();
				runtime()->addInitCalc(calc);
#ifdef RDOSIM_COMPATIBLE
			}
#endif
		}
#ifdef RDOSIM_COMPATIBLE
	}
#endif
}
void RDOParser::runSMRPost()
{
	//! Планирование событий, описанных в SMR
	BOOST_FOREACH(const LPRDOPATPattern& pattern, getPATPatterns())
	{
		LPRDOPatternEvent event = pattern
			.object_dynamic_cast<RDOPatternEvent>();
		if (!event)
			continue;
		rdo::runtime::LPRDOCalc initCalc =
			event->getBeforeStartModelPlaning();
		if (initCalc)
		{
			runtime()->addInitCalc(initCalc);
		}
	}
}
void RDOParser::runRTPPost()
{
	STL_FOR_ALL_CONST(getRTPResTypes(), RTPResTypeIt)
	{
		// Взять очередной тип ресурса в парсере
		LPRDORTPResType pResType = *RTPResTypeIt;

		// Создать соответствующий тип ресурсов в рантайме
		pResType->end();
	}
	}
```

8. Заключение

В рамках данного курсового проекта были получены следующие результаты:

1. Проведено предпроектное исследование системы имитационного моделирования РДО, разобрана теория по проектированию компиляторов, и определены основные направления, проработка которых необходима для реализации задачи построения двухпроходной компиляции.
2. На этапе концептуального проектирования системы выделены подзадачи и определены детали реализации, необходимые для достижения основной цели. Составлено техническое задание.
3. На этапе технического проектирования проработаны изменения в архитектуре системы, изображенные с помощью диаграммы классов в нотации UML. Составлена синтаксическая диаграмма верхнего уровня компилятора РДО.
4. На этапе рабочего проектирования на интерпретируемом языке Python был написан скрипт для генерации файлов из обобщенного файла грамматик, а также скрипт запуска генератора компиляторов bison. Запуск этих скриптов был также интегрирован в систему автоматизации сборки программного обеспечения CMake.

Данные изменения в будущем позволят реализовывать более сложные архитектурные концепции, связанные как с самим синтаксисом языка РДО, так и с введением новых подходов в имитационном моделировании.

9. Список используемых источников

1. Справка по языку РДО [http://rdo.rk9.bmstu.ru/help/];
2. Справка по RAO-studio [http://rdo.rk9.bmstu.ru/help/];
3. Емельянов В. В., Ясиновский С. И. Имитационное моделирование систем: Учеб. Пособие – М.: Издательство МГТУ им. Н. Э. Баумана, 2009. – 584 с.: ил. (Информатика в техническом университете);
4. Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. - М.: ООО "И.Д. Вильямс", 2008. - 1184 стр.;
5. Многопроходный компилятор (англ.) [http://en.wikipedia.org/wiki/Multi-pass_compiler];
6. Мартин Р. Чистый код. Создание, анализ и рефакторинг / пер. с анг. Е. Матвеев – СПб.: Питер, 2010. – 464 стр.;
7. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению. ГОСТ 19.201-78;
8. Б. Страуструп Язык программирования С++. Специальное издание / пер. с англ. – М.: ООО «»Бином-Пресс, 2006. – 1104 с.: ил.;
9. Леоненков. Самоучитель по UML [http://khpi-iip.mipk.kharkiv.edu/library/case/leon/index.html];
10. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. ГОСТ 19.701-90. Условные обозначения и правила выполнения.

Список использованного программного обеспечения

1. RAO-Studio;
2. Python v3.3.2;
3. Sublime Text 2;
4. Notepad++;
5. Autodesk AutoCAD 2012;
6. Microsoft Office Word 2010;
7. Microsoft Office PowerPoint 2010;
8. Microsoft Office Visio 2010;
9. Microsoft Visual Studio 2008.
10. Inkscape 0.48.2;
11. TortoiseSVN;

10. Приложение А. Код скрипта split-bison

```
#!Python3
import sys, re, os, argparse, subprocess

toolname = "split-bison"
codepage = "utf-8"

def divide(expr):
    f1 = expr
    f2 = expr

    # isle PASS1 blocks if file 1
    while f1.count("#PASS1") > 0:
        pos = f1.find("{", f1.find("#PASS1"))
        brackets = 1
        posend = pos
        while brackets > 0:
            posend += 1
            if f1[posend] == '{':
                brackets += 1
            if f1[posend] == '}':
                brackets -= 1
        f1 = f1[: posend] +       "/* -------------- END PASS 1 ------------- */" +\
                                                                    f1[posend + 1 :]
        f1 = f1[: pos] + f1[pos + 1 :]
        f1 = f1.replace("#PASS1", "/* ---------- COMPILER 1st PASS ---------- */", 1)
    # clean PASS2 blocks from file 1
    while f1.count("#PASS2") > 0:
        Ppos = f1.find("#PASS2")
        pos = f1.find("{", Ppos)
        brackets = 1
        posend = pos
        while brackets > 0:
            posend += 1
            if f1[posend] == '{':
                brackets += 1
            if f1[posend] == '}':
                brackets -= 1
        cut = f1[Ppos + 1 : posend]
        f1 = f1[: Ppos] + "\n" * cut.count("\n") + f1[posend + 1 :]

    # isle PASS2 blocks if file 2
    while f2.count("#PASS2") > 0:
        pos = f2.find("{", f2.find("#PASS2"))
        brackets = 1
        posend = pos
        while brackets > 0:
            posend += 1
            if f2[posend] == '{':
                brackets += 1
            if f2[posend] == '}':
                brackets -= 1
```

```
        f2 = f2[: posend] +       "/* -------------- END PASS 2 ------------- */" +
f2[posend + 1 :]
        f2 = f2[: pos] + f2[pos + 1 :]
        f2 = f2.replace("#PASS2", "/* ---------- COMPILER 2st PASS ---------- */", 1)
    # clean PASS1 blocks from file 2
    while f2.count("#PASS1") > 0:
        Ppos = f2.find("#PASS1")
        pos = f2.find("{", Ppos)
        brackets = 1
        posend = pos
        while brackets > 0:
            posend += 1
            if f2[posend] == '{':
                brackets += 1
            if f2[posend] == '}':
                brackets -= 1
        cut = f2[Ppos + 1 : posend]
        f2 = f2[: Ppos] + "\n" * cut.count("\n") + f2[posend + 1 :]

    return [f1,f2]

def main():
    parser = argparse.ArgumentParser(usage = argparse.SUPPRESS, description =\
             "split multipass compiler yx file into a pair of bison grammar files")

    parser.add_argument('inputFile', type = str, help = ".yx input file")
    parser.add_argument('-y1', type = str, default = '', help =\
                        "1st output y file", required = True)
    parser.add_argument('-y2', type = str, default = '', help =\
                        "2nd output y file", required = True)

    args = parser.parse_args()

    inf = open(args.inputFile,"r", encoding = codepage)
    print(toolname + ": parsing " + args.inputFile)

    out1 = open(args.y1, 'w', encoding = codepage)
    out1.truncate()

    out2 = open(args.y2, 'w', encoding = codepage)
    out2.truncate()

    gram = divide(inf.read())

    out1.write(gram[0])
    out2.write(gram[1])

    print(toolname + ": generated " + args.y1 + ", " + args.y2)

    sys.exit(0)

if __name__ == '__main__':
    main()
```

11. Приложение Б. Код скрипта run-bison

```
 #!Python3
import subprocess, os, argparse, re, sys, difflib

def run_bison(yxPath, yPath, cppPath, name, bison, defines):
    procname = os.path.abspath(bison)+ (" --defines=\"" + defines +\
                   "\" ")*(len(defines) and True) + " -p" + name + " \"" + yPath +\
                                                          "\" -o\"" + cppPath + "\""

    output, errr = subprocess.Popen(procname,stdout = subprocess.PIPE,\
                                stderr = subprocess.PIPE, shell = True).communicate()

    errr = errr.decode(sys.stdout.encoding, 'ignore')

    errr = errr.replace(yPath,yxPath)

    if sys.platform == 'win32':
        yPath = yPath.replace("\\","\\\\")
        yxPath = yxPath.replace("\\","\\\\")
        errr = errr.replace(yPath,yxPath)

    if not errr.count("error"):
        if os.path.exists(cppPath):
            strGRAM = open(cppPath, encoding = codepage).read()
            fin = open(cppPath, 'w', encoding = codepage)
            fin.write( strGRAM.replace(yPath,yxPath) )
            fin.close()

    if sys.platform == 'win32':
        return re.sub(r"^(.*)(\.yx)(:)([0-9]*)\.([0-9]*)(-[0-9]*(\.[0-9]*)?)?(:.*)$",\
                                    r"\1\2(\4)\8",    errr, flags=re.MULTILINE)
    else:
        return re.sub(r"^(.*)(\.yx)(:)([0-9]*)\.([0-9]*)(-[0-9]*(\.[0-9]*)?)?(:.*)$",\
                                    r"\1\2\3\4:\5\8", errr, flags=re.MULTILINE)

def main():
    parser = argparse.ArgumentParser(usage = argparse.SUPPRESS, description =\
                         "run bison twice for multipass compiler grammar files")

    parser.add_argument('inputFile', type = str, help = ".yx input file")
    parser.add_argument('-y1', type = str, default = '',\
                                    help = "1st output y file",   required = True)
    parser.add_argument('-y2', type = str, default = '',\
                                    help = "2nd output y file",   required = True)
    parser.add_argument('-n1', type = str, default = '',\
                                    help = "1st output y name",   required = True)
    parser.add_argument('-n2', type = str, default = '',\
                                    help = "2nd output y name",   required = True)
    parser.add_argument('-o1', type = str, default = '',\
                                    help = "1st output cpp file", required = True)
    parser.add_argument('-o2', type = str, default = '',\
                                    help = "2nd output cpp file", required = True)
```

```
parser.add_argument('-bison', type = str, default = '',\
                                    help = "bison path", required = True)
parser.add_argument('-defines', type = str, default = '',\
                                    help = "bison defines")
args = parser.parse_args()

defines = os.path.abspath(args.defines)

print(toolname + ": " + "Executing bison...", file=sys.stderr)

if len(defines) > 0:
    print(toolname + ": " + "defines path: " +defines, file=sys.stderr)

cppPath = os.path.abspath(args.o1)
if os.path.exists(cppPath):
    os.remove(cppPath)
yPath   = os.path.abspath(args.y1)
yxPath  = os.path.abspath(args.inputFile)
print(toolname + ": " + "parsing " + yPath, file=sys.stderr)

out1 = run_bison(yxPath, yPath, cppPath, args.n1, args.bison, defines)

cppPath = os.path.abspath(args.o2)
if os.path.exists(cppPath):
    os.remove(cppPath)
yPath   = os.path.abspath(args.y2)
yxPath  = os.path.abspath(args.inputFile)
print(toolname + ": " + "parsing " + yPath, file=sys.stderr)

out2 = run_bison(yxPath, yPath, cppPath, args.n2, args.bison, defines)

if out1 == out2:
    print(toolname + ": " + "bison output:", file=sys.stderr)
    con = out1.find(": conflicts:")
    if con > -1:

        file_conflict = open(cppPath.replace(".cpp", ".output"), 'r',\
                                                            encoding = codepage)

        conflicts = file_conflict.read()
        conf_pos = conflicts[:conflicts.find("State ")].count("\n") + 1

        if sys.platform == 'win32':
            out1 = out1.replace(": conflicts:",\
                                "(" + str(conf_pos) + ") : warning C0000:")
        else:
            out1 = out1.replace(": conflicts:", ":" + str(conf_pos) + ": warning:")
        conend = out1[:con].rfind("\n")
        if conend == -1:
            conend = 0
        out1 = out1[:conend] + out1[conend:con].replace(yxPath,\
                        cppPath.replace(".cpp", ".output"), 1) + out1[con:]
```

```
        if sys.platform == 'win32':
            out1 = out1.replace(": error:"," : error C0000:")
            out1 = out1.replace(": warning:"," : warning C0000:")
            out1 = out1.replace("\\\\","\\")
        print(out1, file=sys.stderr)
        sys.exit(0)
    else:
        text1_lines = out1.split('\n')
        text2_lines = out2.split('\n')

        diff1 = difflib.unified_diff(text1_lines, text2_lines, lineterm='')

        print("Difference in bison output:")
        for d in list(diff1):
            if (d[0] == '+' or d[0] == '-') and d[1] != '+' and d[1] != '-':
                print(d[0] + '\n' + d[1:])
            else:
                print(d)

        sys.exit(yxPath +\
                   ": error : bison console output doesn't match for grammar files")

toolname = "run-bison"
codepage = "utf-8"

if __name__ == '__main__':
    main()
```

12. Приложение В. Код модуля split-bison.cmake

```
#=============================================================================
# Copyright (c) 2013 Alexandrovsky Kirill <k.alexandrovsky@gmail.com>
#=============================================================================

MACRO(RDO_SPLIT_BISON_TARGET INPUT_FILE Y_1 Y_2)

	MAKE_DIRECTORY(${GRAMMA_OUTPUT_PATH})

	SET(${INPUT_FILE}_INPUT ${GRAMMA_INPUT_PATH}/rdo${INPUT_FILE}.yx)

	SET(BISON_${FILE_TYPE}_INPUT_1 ${GRAMMA_OUTPUT_PATH}/rdo${Y_1}.y)
	SET(BISON_${FILE_TYPE}_INPUT_2 ${GRAMMA_OUTPUT_PATH}/rdo${Y_2}.y)

	SET(${INPUT_FILE}_PASS1 ${GRAMMA_INPUT_PATH}/rdo${Y_1}.y)
	SET(${INPUT_FILE}_PASS2 ${GRAMMA_INPUT_PATH}/rdo${Y_2}.y)

	SET(CPP_OUTPUT_FILE_1 ${GRAMMA_OUTPUT_PATH}/rdogram${Y_1}.cpp)
	SET(CPP_OUTPUT_FILE_2 ${GRAMMA_OUTPUT_PATH}/rdogram${Y_2}.cpp)

	SET(BISON_${INPUT_FILE}_OUTPUTS_1 ${CPP_OUTPUT_FILE_1})
	SET(BISON_${INPUT_FILE}_OUTPUTS_2 ${CPP_OUTPUT_FILE_2})

	SET(OUTPUT_FILE ${CPP_OUTPUT_FILE_1})
	LIST(APPEND OUTPUT_FILE ${CPP_OUTPUT_FILE_2})

	LIST(APPEND OUTPUT_FILE ${GRAMMA_OUTPUT_PATH}/rdogram${Y_1}.dot)
	LIST(APPEND OUTPUT_FILE ${GRAMMA_OUTPUT_PATH}/rdogram${Y_1}.output)

	LIST(APPEND OUTPUT_FILE ${GRAMMA_OUTPUT_PATH}/rdogram${Y_2}.dot)
	LIST(APPEND OUTPUT_FILE ${GRAMMA_OUTPUT_PATH}/rdogram${Y_2}.output)

	SET(BISON_DEFINE)
	SET(BISON_${INPUT_FILE}_OUTPUT_HEADER)
	IF(${ARGC} GREATER 4 OR ${ARGC} EQUAL 4)
		SET(BISON_${INPUT_FILE}_OUTPUT_HEADER ${GRAMMA_H})
		SET(BISON_DEFINES -defines)
		SET(BISON_DEFINE ${BISON_${INPUT_FILE}_OUTPUT_HEADER})
		LIST(APPEND OUTPUT_FILE ${BISON_${INPUT_FILE}_OUTPUT_HEADER})
	ENDIF()

	IF(MSVC)
		ADD_CUSTOM_COMMAND(
			OUTPUT ${OUTPUT_FILE}
			COMMAND set CYGWIN=nodosfilewarning
			COMMAND set BISON_PKGDATADIR=${BISON_FLEX_DIRECTORY}/share/bison
			COMMAND ${PYTHON_EXECUTABLE}
${PROJECT_SOURCE_DIR}/scripts/Python/split-bison.py ARGS ${${INPUT_FILE}_INPUT} -y1
${BISON_${FILE_TYPE}_INPUT_1} -y2 ${BISON_${FILE_TYPE}_INPUT_2}
			COMMAND ${PYTHON_EXECUTABLE}
${PROJECT_SOURCE_DIR}/scripts/Python/run-bison.py ARGS ${${INPUT_FILE}_INPUT} -y1
${BISON_${FILE_TYPE}_INPUT_1} -y2 ${BISON_${FILE_TYPE}_INPUT_2} -o1
```

```
${CPP_OUTPUT_FILE_1} -o2 ${CPP_OUTPUT_FILE_2} -n1 ${Y_1} -n2 ${Y_2} -bison
${BISON_EXECUTABLE}\ -g\ -v ${BISON_DEFINES} ${BISON_DEFINE}
                    DEPENDS ${${INPUT_FILE}_INPUT}
                    COMMENT "[BISON][rdo${INPUT_FILE}] Building parser with bison
${BISON_VERSION}"
                    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
             )
      ELSE()
             ADD_CUSTOM_COMMAND(
                    OUTPUT ${OUTPUT_FILE}
                    COMMAND ${PYTHON_EXECUTABLE}
${PROJECT_SOURCE_DIR}/scripts/Python/split-bison.py ARGS ${${INPUT_FILE}_INPUT} -y1
${BISON_${FILE_TYPE}_INPUT_1} -y2 ${BISON_${FILE_TYPE}_INPUT_2}
                    COMMAND ${PYTHON_EXECUTABLE}
${PROJECT_SOURCE_DIR}/scripts/Python/run-bison.py ARGS ${${INPUT_FILE}_INPUT} -y1
${BISON_${FILE_TYPE}_INPUT_1} -y2 ${BISON_${FILE_TYPE}_INPUT_2} -o1
${CPP_OUTPUT_FILE_1} -o2 ${CPP_OUTPUT_FILE_2} -n1 ${Y_1} -n2 ${Y_2} -bison
${BISON_EXECUTABLE}\ -g\ -v ${BISON_DEFINES} ${BISON_DEFINE}
                    DEPENDS ${${INPUT_FILE}_INPUT}
                    COMMENT "[BISON][rdo${INPUT_FILE}] Building parser with bison
${BISON_VERSION}"
                    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
             )
      ENDIF()
ENDMACRO(RDO_SPLIT_BISON_TARGET)
```