Пример использования Ruby для генерации C++ кода в библиотеке разбора EMI протокола

Евгений Охотников
http://www.intervale.ru

2008.03.05

Аннотация

Данная статья описывает опыт создания генератора C++ кода на Ruby, который позволил упростить реализацию C++ библиотеки для разбора и формирования пакетов транспортного протокола EMI.

Содержание

1 Введение

В одном из проектов потребовалось реализовать на языке C++ библиотеку разбора и формирования пакетов (PDU — Protocol Data Unit) транспортного протокола EMI [1]. В процессе работы над ней выяснилось, что протокол EMI обладает некоторыми особенностями, которые выливаются в большой объем ручного кодирования: в коде образовывалось много очень похожих фрагментов, отличающихся лишь незначительными деталями. Чтобы упростить разработку вместо ручного программирования операций разбора элементов PDU был использован генератор C++ кода, написанный на языке Ruby [2].

Ruby-кодогенератор позволил описывать PDU в декларативной форме. Из декларативного описания генерировалось два C++ файла: заголовочный и файл реализации. В этих файлах находился код уже готового класса для конкретного типа PDU.

Благодаря такому подходу удалось значительно снизить объем ручного кодирования при реализации библиотеки, получившей название *emi_pdu*.

2 Краткий обзор особенностей протокола EMI

2.1 Что такое EMI?

Протокол EMI (External Machine Interface) — это прикладной протокол, который используется рядом операторов сотовой связи для предоставления доступа к своим SMS-центрам. Данный протокол используется для отсылки и получения коротких сообщений.

Протокол EMI базируется на основе протокола ERMES UCP (Universal Computer Protocol) [3].

2.2 Структура и формат EMI PDU

Протокол EMI использует текстовый формат для представления PDU. Все PDU в EMI оформляются в виде последовательности:

```
stx / header / data_fields / checksum etx
```

где *stx* и *etx* — это специальные символы терминаторы с ASCII кодами 0x02 и 0x03 соответственно.

Заголовок PDU (обозначенный выше как *header*) имеет одинаковую для всех PDU структуру:

Название	**Тип**	**Описание**
TRN	2 цифры	Порядковый номер транзакции
LEN	5 цифр	Общее количество символов между stx и etx. Выровненное вправо с лидирующими нулями
O/R	символ ‘O’ или ‘R’	‘O’ означает команду, а ‘R’ — ответ на нее
OT	2 цифры	Тип команды

После заголовка в PDU следует набор полей (обозначенный выше как *data_fields*), относящихся к конкретному типу PDU.

Замыкают PDU два символа контрольной суммы (обозначенные выше как *checksum* и символ терминатор *etx*.

Вот пример EMI PDU (обычно символы *stx* и *etx* в примерах опускаются):

```
49/00046/R/51/A//0031612345678:281102085030/DF
```

Как уже было сказано, протокол EMI текстовый. Это означает, что все числовые данные передаются в нем в виде строковых представлений (так, в вышеприведенном примере количество байт 46 представлено в виде строки ‘00046’). А текст кодируется в виде шестнадцатеричного представления каждого символа — т.е. текстовое сообщение ‘hello’ будет представлено строкой ‘68656C6C6F’.

Не все поля в EMI PDU являются обязательными. Значительная часть полей в PDU может быть опущена. Подобные поля выглядят просто как отсутствие значения между разделителями ‘/’. Так в приведенном выше примере между значениями `A` и `0031612345678:281102085030` находится как раз такое поле.

В EMI существует несколько групп команд, разбитых на *серии* (series), например, 30-я серия, 50-я серия, 60-я серия[1]. Внутри одной серии (группы) все PDU имеют одинаковый набор полей. Так, 50-я серия определяет восемь команд, но все команды содержат одни и те же поля — различается только обязательность их присутствия. Так, в команде UCP51 (*submit_short_message*) поле *DSCTS* (метка времени доставки сообщения) не должно иметь значения, зато в команде UCP53 (*delivery_notification*) это поле является обязательным и не может быть пустым.

2.3 Особенности интерпретации значений отдельных полей PDU

Ряд полей в некоторых PDU зависят от значений друг друга. Можно сказать, что такие поля формируют группы, управлять которыми нужно осторожно, чтобы не создать неправильный PDU. Вот два наиболее характерных примера:

1. Текст SMS в UCP51 и UCP52 (*deliver_short_message*) PDU: *MT* (тип сообщения), *NB* (количество бит в сообщении) и *Msg*[2] (сам текст сообщения). Так, если сообщение содержит только 7-ми битовый текст, то поле MT должно получить значение '3', поле NB не должно быть задано. Если же сообщение содержит 8-ми или 16-ти битовый текст, то поле MT должно получить значение '4', а поле NB должно содержать количество бит в тексте сообщения.
2. Номер отправителя SMS задается полями *OAdC* (адрес отправителя сообщения) и *OTOA* (тип адреса отправителя сообщения). Если OAdC содержит т.н. алфавитно-цифровой адрес (например, alpha@num), то значение OAdC специальным образом кодируется, а OTOA получает значение '5039'.

2.4 В чем сложность работы с EMI PDU?

Основная сложность при работе с EMI PDU состоит в том, что EMI налагает строгие ограничения на размер и формат всех полей в PDU. Например, поле *LEN*:

- имеет длину 5 символов;
- может содержать только цифры, поскольку предназначено для хранения числа байт в PDU;
- если число байт в PDU меньше пятизначного значения (например, 78), то поле *LEN* должно содержать лидирующие нули, т.е. иметь значение 00078.

Подобные ограничения нужно учитывать как при формировании PDU, так и при разборе PDU. И, если при разборе выясняется, что какое-то поле PDU не соответствует описанному в спецификации EMI формату, то все PDU должно быть отвергнуто как некорректное.

Поэтому при реализации работы с полями PDU приходится писать много кода, отвечающего за контроль корректности значения PDU. Т.е. сложность проистекает не из сложности самого протокола, а из большого количества мелких деталей, которые нужно учитывать при работе с PDU.

3 Возможные подходы к работе с EMI PDU

В процессе работы над emi_pdu рассматривались два основных подхода к организации работы с PDU:

[1] Название серии определяет, какие значения может принимать поле *OT* в заголовке PDU для всех PDU этой серии. Так, в 30-й серии поле *OT* имеет значение 3x (например, 31 или 33, но не 41 или 52).

[2] В спецификации EMI данное поле имеет три названия (NMsg, AMsg и TMsg), в зависимости от типа данных в тексте SMS.

1. Представление PDU в виде списка строковых значений полей плюс набор операций для получения этого списка из готового PDU и формирование готового PDU по списку значений. Похожий подход используется в OpenSource проекте Kannel [4]. На C++ он мог бы быть выражен чем-то вроде:

```
class pdu_fields_t
  {
  public :
    ...
    const std::string &
    query_field( int field_id ) const;

    void
    set_field( int field_id, const std::string & value );

    bool
    is_field_present( int field_id ) const;
    ...
  };

std::auto_ptr< pdu_fields_t >
parse_pdu( const std::string & buffer );

std::string
make_pdu( const pdu_fields_t & fields );
```

   Работа с таким классом могла бы выглядеть следующим образом:

```
// Фрагмент формирования UCP51 PDU.
pdu_fields_t pdu;
...
pdu.set_field( UCP5X_MT, "4" );
pdu.set_field( UCP5X_NB, itoa( body.length() * 8 ) );
pdu.set_field( UCP5X_Msg, body );
...
// Фрагмент разбора UCP52 PDU.
if( "4" == pdu.query_field( UCP5X_MT ) )
  {
    if( !pdu.is_field_present( UCP5X_NB ) )
      throw invalid_pdu( "NB not present" );
    int byte_count = atoi(pdu.query_field(UCP5X_Msg).c_str()) / 8;
    ...
  }
```

2. Создание классов для различных типов PDU, которые бы скрывали от пользователя детали формата отдельных полей и предоставляли методы для управления конкретными полями. Например, для работы с UCP51 PDU существовал бы класс:

```
class ucp51_pdu_t
  {
  public :
    ...
    int query_MT() const;
    void set_MT( int value );
```

```
7     bool is_MT_present() const;
8
9     int query_NB() const;
10    void set_NB( int value );
11    bool is_NB_present() const;
12
13    const std::string & query_Msg() const;
14    void set_Msg( const std::string & value );
15    bool is_Msg_present() const;
16    ...
17  };
```

И работа с подобными классами могла бы выглядеть как:

```
1  ucp51_pdu_t pdu;
2  ...
3  pdu.set_MT( 4 );
4  pdu.set_NB( body.length() * 8 );
5  pdu.set_Msg( body );
6  ...
```

Первый подход обладает следующими недостатками:

- существенно увеличивается объем кода, который приходится писать при формировании и разборе PDU;
- при формировании/разборе PDU приходится учитывать конкретный формат значения конкретного поля (например, поля *TRN* и *LEN* в заголовке PDU должны содержать лидирующие нули, а тело сообщения в *Msg* должно быть представлено в шестнадцатеричном виде);
- низкий уровень контроля за корректностью работы с PDU со стороны компилятора. Так, поля PDU идентифицируются целочисленными константами и нет никаких препятствий к тому, чтобы при формировании одного типа PDU случайно не использовать идентификатор поля из другого типа PDU (например, применить константу `UCP6X_OAdC` вместо `UCP5X_OAdC` при формировании UCP51 PDU).

Второй подход, когда для каждого типа PDU создается свой C++ класс, а для каждого поля в PDU — свой набор методов, устраняет эти недостатки:

- объем кода уменьшается, т.к. часть действий (например, преобразование форматов) выполняется внутри методов C++ классов PDU;
- уменьшается количество мест, где можно допустить случайную ошибку: за формат отвечает сам класс PDU, а компилятор гарантирует, что у PDU не будет установлено поле, которого в PDU нет вообще (т.к. для этого поля нет методов в классе PDU).

В библиотеке emi_pdu C++ классы создаются не на каждый тип PDU, а на каждую серию PDU. Т.е. нет классов `ucp51_pdu_t`, `ucp52_pdu_response_t` и т.д. Вместо этого есть классы `operation_t` и `response_t` для каждой серии PDU, помещенные с соответствующие пространства имен (т.е. `ucp5x::operation_t`, `ucp6x::response_t` и др.). Сделано это для уменьшения объема библиотеки, т.к. внутри каждой серии все PDU имеют одинаковую структуру. Поэтому достаточно иметь один класс для всей серии, а не набор одинаковых классов для каждого отдельного PDU в серии[3].

[3]На принятие такого решения серьезно повлияла специфика самой задачи: нужно было поддерживать всего несколько типов PDU и проводить лишь несколько операций над этими PDU. Если бы

4 Ruby-кодогенерация для разбора EMI

4.1 Кодогенерация: что и как

Зачем в emi_pdu понадобилась кодогенерация? Выбранный подход к работе с EMI PDU требовал создания для каждого поля F набора из следующих методов:

```
// Возвращает true, если поле определено в PDU.
bool is_F_defined() const;

// Сбрасывает значение поля, после чего поле считается неопределенным.
void drop_F();

// Получить значение поля.
// Порождает исключение, если поле не определено.
const T & query_F() const;

// Получить или значение поля, или значение по умолчанию,
// если поле в PDU не определено.
T fetch_F( const T & default_value ) const;

// Установить значение поля.
void set_F( const T & value );
```

где T — это тип поля (например, int или std::string).

Естественно, что написание пяти практически одинаковых методов для каждого поля в 50-й серии PDU (где PDU содержит 33(!) поля) — это слишком утомительно для ручного кодирования. Гораздо лучше с этим справится кодогенератор, которому нужно всего лишь указать имя поля и его тип.

Так же каждый класс для PDU должен содержать реализацию трех виртуальных методов:

```
// Преобразование объекта PDU в соответствующий байтовый поток.
virtual void
encode( oess_1::io::ostream_t & to ) const;

// Извлечение объекта PDU из байтового потока.
virtual void
decode( const unclassified_fields_t & fields );

// Формирование текстового отладочного описания PDU.
virtual void
debug_dump( std::ostream & to ) const;
```

Особенностью методов `encode` и `decode` является то, что поля PDU должны обрабатываться в строго определенном порядке. Поэтому от ручного написания этих методов так же хотелось избавиться, чтобы снизить вероятность разного порядка обработки полей в каждом из этих методов.

Т.е., если построить систему кодогенерации, которая бы брала на вход описание полей PDU (имя поля, его тип и другие параметры) и генерировала бы набор методов для каждого поля плюс реализацию методов `encode`, `decode` и `debug_dump`, то ручное кодирование классов PDU будет сведено к минимуму: в некоторых случаях

задача была более серьезной, то выбранный подход с кодогенерацией позволил бы без особых проблем генерировать большее количество классов — хоть по одному классу на каждый PDU.

достаточно только составить список полей PDU, а в оставшихся случаях, вручную можно создавать только наследников сгенерированных автоматически классов[4].

Именно такая система и была создана при разработке emi_pdu. Программист составлял специальное описание каждой серии PDU, во время компиляции проекта запускался кодогенератор, который брал описание и генерировал два файла — заголовочный файл и файл с реализацией. Эти файлы подключались к исходным текстам библиотеки через `#include` и, таким образом, сгенерированные классы компилировались вместе с остальным кодом библиотеки.

4.2 Ruby: почему и зачем

Итак, кодогенератор нужен для того, чтобы взять некоторый список полей и построить два выходных файла. Оставалось решить два вопроса:

1. В каком формате должен быть представлен этот список?
2. На чем будет написан сам кодогенератор?

Вопрос выбора формата описания полей PDU является вопросом выбора DSL (Domain Specific Language) [5], т.е. специализированного мини-языка для данной предметной области. Можно было остановиться на каком-то из внешних (т.н. external DSL) языков, вроде XML или YAML. А можно было использовать внутренний (т.н. internal DSL) язык, скажем, макросы C/C++. Но здесь свою роль сыграл второй вопрос — реализация самого кодогенератора.

Можно было бы использовать построенный на основе XML DSL и применить для генерации XSLT преобразования. Но это бы означало подключение к проекту инструментов, которые ранее для проекта не использовались. Кроме того, для подобного решения необходимо было иметь опыт использования связки XML+XSLT, которого у автора не было, как и не было времени на изучение возможностей данной связки.

Кодогенератор можно было бы написать на C++. Т.е. в процессе построения проекта сначала строится специальная утилита, которая берет описания PDU и генерирует результирующие файлы. Но предыдущий опыт создания таких утилит-генераторов на C++ показал, что данная задача **гораздо проще** решается на динамических, скриптовых языках вроде Ruby, Python или Perl. Поэтому вариант, когда генератор C++ кода пишется на одном из этих языков выглядел наиболее привлекательным.

Свою роль в данном случае сыграло и то, что автор имел предшествующий опыт создания кодогенераторов на Ruby, а в проекте emi_pdu использовалась написанная же на Ruby система сборки *Mxx_ru* [6]. Поэтому интегрировать Ruby-кодогенератор в процесс компиляции emi_pdu не представляло никакой сложности.

Раз Ruby используется для кодогенерации, то чтобы упростить создание кодогенератора, можно было оказаться от использования внешнего DSL в пользу внутреннего DSL на языке Ruby: в этом случае в кодогенераторе отсутствует такая фаза, как чтение и разбор описания в каком-то внешнем представлении. Т.е. при использовании Ruby нет надобности делать описание на XML или YAML, загружать это описание, как-то его обрабатывать и уже затем генерировать C++ код. В Ruby описание PDU можно было сделать в виде небольшой Ruby программы, которая бы запускалась и сразу же генерировала C++ код, без дополнительных операций.

[4] В emi_pdu на этом построен учет особенностей взаимосвязанных полей (т.к. *MT*, *NB* и *Msg*) — сгенерированный класс предоставляет только базовый функционал, а написанный вручную наследник добавляет к нему несколько еще более высокоуровневых методов для сокрытия от пользователя деталей взаимосвязей некоторых полей.

4.3 Как организован кодогенератор

Кодогенерация в проекте emi_pdu реализована с помощью Ruby-библиотеки *RuCodeGen* [7]. Данная библиотека была разработана автором статьи специально для подобных целей. RuCodeGen предоставляет программисту базовую функциональность для написания кодогенератора. При использовании RuCodeGen требуется создать свои классы для представления DSL и формирования результирующего исходного кода. Загрузку DSL, контроль необходимости перегенерации и запуск генерации выполняет RuCodeGen.

RuCodeGen поддерживает два способа загрузки DSL — загрузка DSL из отдельных Ruby-файлов и загрузка DSL, встроенного в исходный файл на другом языке программирования. В первом случае DSL оформляется в виде самостоятельной Ruby-программы, например:

```
1 require 'emi_pdu_1/pdu_generator'
2
3 pdu_class :operation_t do
4   ...
5 end
```

которая запускается на выполнение обычным образом, с помощью Ruby-интерпретатора:

```
ruby pdu_description.rb
```

Во втором случае DSL помещается внутри комментария в C++ном файле, и обрамляется специальными маркерами:

```
1 #if 0 /* RuCodeGen::Embedded::begin */
2 require 'emi_pdu_1/pdu_generator'
3
4 pdu_class :operation_t do |pdu|
5   ...
6 end
7 #endif /* RuCodeGen::Embedded::end */
```

Такое описание называется встроенным. Запустить по нему кодогенерацию несколько сложнее, но зато оно позволяет поместить описание PDU непосредственно в заголовочный файл.

При разработке кодогенератора под RuCodeGen нет различий, будет ли этот кодогенератор брать описания из отдельных Ruby-файлов, или же описание будет встроенным в C++ файл — обработкой этих ситуаций занимается сам RuCodeGen.

Кодогенератор для emi_pdu включает в себя: метод `pdu_class` (который как раз и реализует встроенный DSL) и два класса-генератора, которые берут результаты работы `pdu_class` и строят необходимый C++ код (один генератор используется для заголовочных файлов, второй — файлов реализации), а так же несколько вспомогательных классов.

4.4 Как создаются описания PDU

Для описания PDU пользователь должен вызвать метод `pdu_class`, в котором необходимо перечислить все поля PDU. Вот небольшой пример:

```
1 pdu_class :negative_result_t do |pdu|
2   pdu.decl_file :script_relative => 'ucp3x.negative_result.hpp.inl'
3   pdu.impl_file :script_relative => '../ucp3x.negative_result.cpp.inl'
```

```
4
5     pdu.field :NAcK, any_char( 1 ), :presence => :mandatory,
6         :default_value => 'nack_value'
7     pdu.field :EC, left_padded_uint( 2 ), :presence => :mandatory
8     pdu.field :SM, any_char( 128 )
9   end
```

В первой строке вызывается метод `pdu_class`, которому в качестве имени передается имя результирующего класса. В данном случае это `negative_result_t`. Имя класса, как и ряд других аргументов, задаются в виде т.н. символов (symbol в терминологии Ruby) — это аналог строк, но с некоторыми отличиями. Записывается символ в виде лидирующего двоеточия, за которым следует идентификатор[5]. В Ruby принято использовать символы для обозначения сущностей, играющих роль идентификаторов (например, имен переменных, констант и т.д.).

При вызове метода в Ruby скобки вокруг параметров метода не обязательны, поэтому конструкции:

```
1   pdu_class :negative_result_t do |pdu|
2     pdu.decl_file :script_relative => 'ucp3x.negative_result.hpp.inl'
3     pdu.impl_file :script_relative => '../ucp3x.negative_result.cpp.inl'
```

полностью эквивалентны следующему варианту со скобками:

```
1   pdu_class( :negative_result_t ) do |pdu|
2     pdu.decl_file( :script_relative => 'ucp3x.negative_result.hpp.inl' )
3     pdu.impl_file( :script_relative => '../ucp3x.negative_result.cpp.inl' )
```

Однако, запись без скобок выглядит более декларативной, менее похожей на обычную программу. Эта особенность языка Ruby является одним из факторов, которые делают Ruby удобным для реализации встроенных DSL.

Конструкция `do...end` обозначает начало и конец блока кода, который передается дополнительным аргументом в метод `pdu_class`. Блок кода в Ruby — это аналог анонимных функций или делегатов в других языках программирования. При реализации встроенных DSL на Ruby блоки кода очень широко используются, т.к. они позволяют создавать видимость вложенных описаний. В данном примере блок кода содержит инструкции, которые описывают состав полей PDU для работы с которым предназначен класс `negative_result_t`. Блок кода получает один аргумент — `pdu`. Это экземпляр вспомогательного класса (являющегося частью описываемого здесь кодогенератора) для накопления описаний полей PDU.

Строки 2-3 содержат инструкции для указания имен результирующих файлов. Метод `pdu.decl_file` задает имя заголовочного файла, а `pdu.impl_file` — имя файла реализации. Данные методы получают свои аргументы в виде хэш-таблицы, которая автоматически строится Ruby-интерпретатором, когда он видит конструкцию `A => B` (где A — это ключ, а B — значение). В данном примере метод `pdu.decl_file` получает хэш-таблицу, в которой есть всего один элемент, ключом которого является символ `:script_relative`, а значением — строка с именем файла. В строках 5-6 так же встречается хэш-таблица, но уже с двумя элементами.

Использование хэш-таблиц при передаче аргументов методов является распространенной практикой в Ruby. С их помощью в Ruby эмулируется такая возможность некоторых языков программирования (например, Ada и Nice), как именованные аргументы.

[5]Вместо идентификатора так же может использоваться заключенная в кавычки строка.

Строки 5-8 содержат описание трех полей PDU с именами *NAcK*, *EC* и *SM*. Каждое поле описывается с помощью вызова метода `pdu.field`, который получает в качестве аргументов имя поля, тип и формат поля[6], а так же набор необязательных параметров для уточнения характеристик поля. Так, поле *NAcK* представляется в PDU одним произвольным символом, поле *EC* должно содержать только две цифры, а поле *SM* может быть произвольной строкой, но длинной не более 128 символов. В C++ программе же поле *EC* будет представлено беззнаковым целым, которое должно быть в диапазоне $[0, 99]$. Причем, если значение *EC* будет меньше 10, то оно будет записано в PDU с лидирующим нулем.

Кроме того, поля *NAcK* и *EC* помечены как обязательные. Для таких полей будет сгенерирован код, который будет проверять наличие значений таких полей при сериализации/десериализации PDU. И, если значение поля отсутствует, будут порождаться соответствующие исключения.

Для поля *NAcK* так же задано начальное значение. Т.е. при формировании такого PDU в программе не нужно будет задавать значение *NAcK*, т.к. вспомогательный код будет содержать соответствующую инициализацию:

```
1  negative_result_t::negative_result_t()
2    : m_NAcK( "NAcK", nack_value )
3    , m_EC( "EC" )
4    , m_SM( "SM" )
5    {}
```

4.5 Подключение кодогенератора в C++ проект

Для того, чтобы использовать кодогенератор в emi_pdu программисту требуется выполнить следующие действия:

1. Описать конкретный тип PDU в виде встроенного в C++ный заголовочный файл фрагмента Ruby кода. Например:

```
1  #if !defined( EMI_PDU_1__UCP3X_HPP )
2  #define EMI_PDU_1__UCP3X_HPP
3  ...
4  #if 0 /* RuCodeGen::Embedded::begin */
5  require 'emi_pdu_1/pdu_generator'
6
7  pdu_class :operation_t do |pdu|
8    ...
9  end
10 #endif /* RuCodeGen::Embedded::end */
11 ...
12 #endif
```

2. Подключить в соответствующе C++ файлы результаты кодогенерации через директивы `#include`. Например, в заголовочный файл:

```
1  // Заголовочный файл со встроенным DSL.
2  ...
3  #if 0 /* RuCodeGen::Embedded::begin */
4  require 'emi_pdu_1/pdu_generator'
5
```

[6] Тип и формат поля задается функциями `any_char`, `left_padded_uint` и др. Эти функции являются частью генератора для EMI.

```
pdu_class :operation_t do |pdu|
  pdu.decl_file :script_relative => 'ucp3x.operation.hpp.inl'
  pdu.impl_file :script_relative => 'ucp3x.operation.cpp.inl'
  ... # Описание PDU.
end
#endif /* RuCodeGen::Embedded::end */
// Подключение описание сгенерированного класса operation_t.
#include "ucp3x.operation.hpp.inl"
...
```

и в файл реализации:

```
#include <emi_pdu_1/h/ucp3x.hpp>
...
#include "ucp3x.operation.cpp.inl"
```

3. Указать в проектном файле имена заголовочных файлов, содержащих встроенный DSL с помощью метода `add_embedded`:

```
require 'rubygems'

gem 'Mxx_ru', '>= 1.3.0'

require 'mxx_ru/cpp'
require 'mxx_ru/cpp/rucodegen'

MxxRu::Cpp::lib_target {

  target 'lib/emi_pdu.1.0.0'

  required_prj 'oess_1/io/prj.rb'

  cpp_source 'consts.cpp'
  cpp_source 'common.cpp'
  cpp_source 'encodings.cpp'

  r = generator( MxxRu::Cpp::RuCodeGen.new( self ) )
  r.add_embedded 'h/ucp6x.hpp'
  r.add_embedded 'h/ucp5x.hpp'
  r.add_embedded 'h/ucp3x.hpp'

  cpp_source 'ucp6x.cpp'
  cpp_source 'ucp5x.cpp'
  cpp_source 'ucp3x.cpp'
}
```

Такое описание будет указывать Mxx_ru, что перед началом компиляции проекта emi_pdu нужно запустить RuCodeGen для описаний, встроенных в файлы `h/ucp6x.hpp`, `h/ucp5x.hpp` и `h/ucp3x.hpp`.

4.6 Некоторые детали реализации кодогенератора

Метод `pdu_class`, который может показаться самой важной частью описанного выше DSL, в действительности является очень простым методом-координатором. Вот код `pdu_class`:

```
1  def pdu_class( class_name, &blk )
2    description = EmiProto_1::PduDescription.new( class_name )
3    blk[ description ]
4
5    fail "decl_file must be defined" unless description.get_decl_file
6    fail "impl_file must be defined" unless description.get_impl_file
7
8    RuCodeGen::Generators.add(
9        RuCodeGen::FilenameProducer.produce(
10           $0, description.get_decl_file ),
11       EmiProto_1::DeclGenerator.new( description ) )
12   RuCodeGen::Generators.add(
13       RuCodeGen::FilenameProducer.produce(
14           $0, description.get_impl_file ),
15       EmiProto_1::ImplGenerator.new( description ) )
16
17   description
18 end
```

Метод `pdu_class` отвечает за выполнение следующих основных задач:

1. Создание объекта, который будет содержать описание параметров генерируемого PDU и который будет передаваться единственным аргументом в предоставляемый пользователем блок кода. Для этого в строке 2 создается объект типа *PduDescription*.
2. Вызов пользовательского блока кода с передачей только что созданного объекта типа PduDescription в качестве аргумента (строка 3). Внутри этого блока пользователь вызывает методы `decl_file`, `impl_file` и `field`, в результате чего в экземпляре PduDescription накапливается полное описание PDU.
3. Первичная проверка созданного в пользовательском блоке кода описания (строки 5-6). Так, нельзя запускать генерацию, если пользователь не определил имена результирующих файлов.
4. Создание объектов-генераторов и регистрация их в соответствующей подсистеме RuCodeGen (строки 8-15). При генерации PDU создается два файла (заголовочный и реализации), поэтому `pdu_class` создает два объекта: объект типа *DeclGenerator* строит заголовочный файл, а объект типа *ImplGenerator* — файл реализации.

Класс *PduDescription* в составе кодогенератора так же не является самой сложной частью. Он всего лишь содержит набор методов-setter-ов (т.к. `decl_file`, `impl_file` и `field`), несколько атрибутов для накопления значений и набор методов-getter-ов для извлечения этих значений.

Основную же рабору в кодогенераторе выполняют классы *DeclGenerator* и *ImplGenerator*. На их долю приходится значительная часть объема кодогенератора, хотя по большей части это очень простой код для формирования C++ фрагментов, вроде:

```
1  def make_fields_accessor( fields )
2    fields.inject( "" ) do |result, f|
3      result << <<EOS
4    //! Установлено ли значение #{f.name}.
5    bool
6    is_#{f.name}_defined() const;
```

```
  //! Сброс значения #{f.name}.
  void
  drop_#{f.name}();
  //! Получение значения #{f.name}.
  /*!
   * \\throw ex_t если значение не установлено.
   */
  const #{f.type} &
  query_#{f.name}() const;
  //! Получение значения #{f.name} если оно определено,
  //! или значения по умолчанию в противном случае.
  #{f.type}
  fetch_#{f.name}( const #{f.type} & default_value ) const;
  //! Установка значения #{f.name}.
  void
  set_#{f.name}( const #{f.type} & v );

EOS
    result
  end
end
```

Весь код генератора привести в статье не представляется возможным, но можно схематично представить его основные части:

```
require 'rucodegen'

module EmiProto_1
  # Стандартный набор свойств поля PDU по умолчанию.
  DEFAULT_FIELD_PARAMS = {
      :presence => :optional,
      :visibility => :public }

  # Описание одного поля.
  class Field
    attr_reader :name
    attr_reader :type
    attr_reader :format
    attr_reader :params

    def initialize( name, format, params )
      @name = name
      @type = format.underlying_type
      @format = format.format_type
      @params = DEFAULT_FIELD_PARAMS.merge( params )
    end
    ... # Дополнительные методы...
  end

  # Интерфейс класса, описывающего формат поля.
  class FieldFormat
    # Возвращает имя типа, который должен быть типом поля.
    attr_reader :underlying_type

```

```
30      # Возвращает имя класса-форматера значения поля.
31      attr_reader :format_type
32
33      def initialize( underlying_type, format_type )
34        @underlying_type = underlying_type
35        @format_type = format_type
36      end
37    end
38
39    # Класс с описанием PDU.
40    class PduDescription
41      # Имя результирующего класса.
42      attr_reader :class_name
43
44      # Список полей.
45      attr_reader :fields
46
47      # Имя файла с декларацией класса.
48      def decl_file( name_params )
49        @decl_file = name_params
50      end
51      def get_decl_file
52        @decl_file
53      end
54
55      # Имя файла с реализацией класса.
56      def impl_file( name_params )
57        @impl_file = name_params
58      end
59      def get_impl_file
60        @impl_file
61      end
62
63      def initialize( class_name )
64        @class_name = class_name
65        @fields = []
66      end
67
68      # Добавление очередного поля.
69      def field( field_name, field_format, params = DEFAULT_FIELD_PARAMS )
70        @fields << Field.new( field_name, field_format, params )
71      end
72    end
73
74    # Генератор декларации класса.
75    class DeclGenerator
76      def initialize( pdu_description )
77        @pdu = pdu_description
78      end
79
80      # Реализация генератора.
81      # Генерируемый код помещается в поток to.
82      def generate( to )
83        ...
```

```
84      end
85    end
86
87    # Генератор реализации класса.
88    class ImplGenerator
89      def initialize( pdu_description )
90        @pdu = pdu_description
91      end
92
93      # Реализация генератора.
94      # Генерируемый код помещается в поток to.
95      def generate( to )
96        ...
97      end
98    end
99
100   # Вспомогательные методы для описания формата поля.
101   def left_padded_uint( width )
102     EmiProto_1::FieldFormat.new(
103         "unsigned int",
104         "left_padded_uint_format_t< #{width} >" )
105   end
106   def any_char( max_len )
107     EmiProto_1::FieldFormat.new(
108         "std::string",
109         "any_char_format_t< #{max_len} >" )
110   end
111   ...
112
113 end # module EmiProto_1
114
115 # Сам DSL.
116 def pdu_class( class_name, &blk )
117   ... # Реализация показана выше.
118 end
```

5 Заключение

Описанный выше кодогенератор на момент написания статьи содержит 414 строк. С его помощью генерируется приблизительно 4700 строк C++ кода, в котором определяется 9 типов PDU (в общей сложности 63 поля). И только для одного типа PDU потребовалось создание наследника автоматически сгенерированного класса для сокрытия некоторых деталей работы EMI (объем этого вручную написанного класса почти 400 строк, включая комментарии).

Первая реализация кодогенератора заняла около 4-х часов. Впоследствии кодогенератор был расширен — для каждого поля PDU потребовалось создавать дополнительный метод `fetch_*`, которого не было в первоначальном варианте. Добавление метода `fetch_*` не потребовало изменения ни строчки во встроенных DSL описаниях.

Подобные показатели позволяют считать описанный кодогенератор удачным опытом использования Ruby для генерации C++ кода. Однако, нужно отметить, что часть успеха определялась тем, что применявшаяся система сборки проектов

Mxx_ru имела встроенные средства поддержки генерации кода во время компиляции. Если бы подключение кодогенератора в процесс сборки проекта не был настолько простым, то, вероятно, эффект от использования Ruby-кодогенератора заметно снизился бы. Возможно, в этом случае вместо Ruby пришлось бы использовать штатное средство языка C++ — его препроцессор.

Благодарности. Автор выражает благодарность Сергею Круподерову, Александру Мигай, Игорю Мирончику и Максиму Янченко aka jazzer[7] за помощь в подготовке данной статьи.

Список литературы

[1] Short Message Service Centre 4.6 EMI - UCP Interface Specification – CMG Wireless Data Solutions, 2003

[2] `http://www.ruby-lang.org`

[3] ETSI ETS 300 133-3 Paging Systems (PS); European Radio Messaging System (ERMES) Part 3: Network aspects; Section 9: I5 interface, 1997

[4] `http://www.kannel.org`

[5] `http://martinfowler.com/dslwip`

[6] `http://www.rubyforge.org/projects/mxx-ru`

[7] `http://www.rubyforge.org/projects/rucodegen`

[7] `http://www.rsdn.ru/Users/8211.aspx`