Санкт-Петербургский Государственный Университет
Математико-механический факультет

Средства описания генераторов кода для предметно-ориентированных решений в metaCASE-средстве QReal

Курсовая работа студента 345 группы
Подкопаева Антона Викторовича

Научный руководитель
ст. пр. Т. А. Брыксин

Санкт-Петербург
2012

Оглавление

Введение

На данный момент существует достаточно много систем визуального моделирования — CASE-систем[1]. Некоторые из них являются также metaCASE-системами[2] (Actifsource[3], Microsoft Visual Studio Visualization and Modeling SDK[4], MetaEdit+[5]), что позволяет внутри них разрабатывать новые визуальные языки [3] и редакторы к ним.

Одной из задач, возникающих при создании предметно-ориентированного инструментария внутри metaCASE-систем, является разработка генератора запускаемого программного кода или другого текстового представления данных, необходимого при разработке продукта (документация, прототип приложения, интерфейсы подсистем). Подобные генераторы необходимы практически для каждого визуального языка, таким образом для DSM-подхода[6] становится крайне важно упростить процесс разработки подобных трансляторов.

Хотя логически генераторы достаточно простые, написание их на языке общего назначения превращается в длительный и сложный процесс. Задачей данной работы была разработка системы, которая упростила бы этот процесс.

1 Постановка задачи

В рассматриваемой области генератор для визуальных языков — это инструмент, который обходит граф модели и получает некоторое текстовое представление этого графа. Такие трансляторы по способу их работы можно разделить на две категории: с использованием шаблонов генерации [2] и без. Генераторы без использования шаблонов обычно включают в свой исходный код большое количество управляющих символов системы ввода-вывода (табуляций, переводов строки) для текстового форматирования результата. Написание подобных систем осложнено отсутствием наглядности в представлении итогового кода. Для решения проблемы можно использовать шаблоны генерации, которые фиксируют структуру получаемого результата, оставляя за самим генератором возможность подставлять в необходимые позиции требуемое наполнение.

---

[1] CASE-система, http://en.wikipedia.org/wiki/CASE_tool

[2] metaCASE-система, http://en.wikipedia.org/wiki/MetaCASE_tool

[3] Actifsource http://www.actifsource.com/

[4] Microsoft Visual Studio Visualization and Modeling SDK http://archive.msdn.microsoft.com/vsvmsdk

[5] MetaEdit+ http://www.metacase.com/

[6] Предметно-ориентированное моделирование, (Domain Specific Modeling, DSM), http://en.wikipedia.org/wiki/Domain-specific_modeling

Необходимо было повысить уровень абстракции при создании генераторов, так как слишком много усилий тратится на рутинную работу. При написании “вручную” на языке общего назначения нескольких генераторов было замечено, что большинство из них имели схожие логические блоки команд, например, позволяющие осуществлять навигацию по графу модели или работу с файлами. Было решено сделать специальный язык и выделить часто используемые в генераторах действия в отдельные языковые конструкции. Программы представляли бы из себя шаблоны с управляющими конструкциями, что позволило бы использовать основной плюс шаблонного метода (наглядность), уйдя от написания большого количества рутинного кода, заменяя целые наборы строк, отвечающих за типовые операции, одной-двумя инструкциями.

Подобный подход используются и в индустрии. В средстве MetaEdit+ компании MetaCase применяется язык описания генераторов, который представляет из себя набор команд для перехода между объектами, получения их параметров, передачи таких параметров в результирующий поток [5]. В таком подходе не используется наглядность шаблонного метода, но он гибок, что позволяет писать достаточно широкий класс генераторов. В то время как в продукте Actifsource используется специальный редактор описания именно шаблона генерации [4]. Это вносит существенное ограничение на визуальный язык, заставляя его быть довольно громоздким. Однако, повышается наглядность такого подхода в связке с редактором, который исключает из видимого текста конструкции возвращения свойств объекта, перенося их на особые символы. Например, подчеркнутая конструкция в шаблоне означает, что в этом месте результата должно появится не сама она, а соответствующее поле объекта диаграммы.

Было принято решение совместить достоинства этих двух подходов. В отличие от средств MetaEdit+ на первое место выводятся не управляющие конструкции, а скелет итогового приложения — шаблон. С другой стороны, сохраняются управляющие конструкции, позволяющие уменьшить влияние на визуальный язык.

2 Обзор существующих решений

2.1 Microsoft Visualization and Modeling SDK

Microsoft Visualization and Modeling SDK (VMSDK) — это средство визуального моделирования, которая добавляет в Microsoft Visual Studio возможность разрабатывать предметно-ориентированные решения. Предоставляется как подключаемый модуль к Microsoft Visual Studio SDK.

Для генерации кода используется язык T4[1]. Шаблон T4 представляет из себя комбинацию управляющей логики и текстовых блоков, которые обрабатываются согласно этой логике. Управляющие конструкции выделяются символами <#, #>.
Для описания логики используются Visual Basic и C#. Выбор языка происходит с помощью прямого указания соотвествующей инструкции в шаблоне:

```
<#@ template language="C#" #>
```

Шаблон транслируется в программу на Visual Basic или C#, результатом работы которой является код целевой программной системы.

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ output extension=".cs" #>
<# var properties = new string [] {"P1", "P2", "P3"}; #>
class MyGeneratedClass {
<#
  foreach (string propertyName in properties)
  { #>
  private int <#= propertyName #> = 0;
<# } #>
}
```

Рисунок 1. Пример генератора на языке T4

Доступ к модели осуществляется стандартными средствами языка. Так, в случае C#, диаграмма представляется как набор объектов классов.

[1] T4, http://msdn.microsoft.com/en-us/library/bb126445.aspx

2.2 Actifsource

Actifsource — это дополнение к свободной кроссплатформенной среде разработки Eclipse, предоставляющее широкий спектр возможностей для разработки программного обеспечения: от графического моделирования и генерации кода до рефакторинга диаграмм. Основная часть разрабатывается сообществом и предоставляется бесплатно.

В продукте Actifsource в роли генератора кода также выступает шаблон, который описывается в специальном редакторе. Этот редактор позволяет полностью убрать управляющие конструкции из шаблона за счет анализа вводимого текста. Так, к примеру, если при написании Service.name воспользоваться автодополнением, то эта подстрока будет выделена подчеркиванием, а при генерации будет заменена на соответсвующим значением для элемента диаграммы.

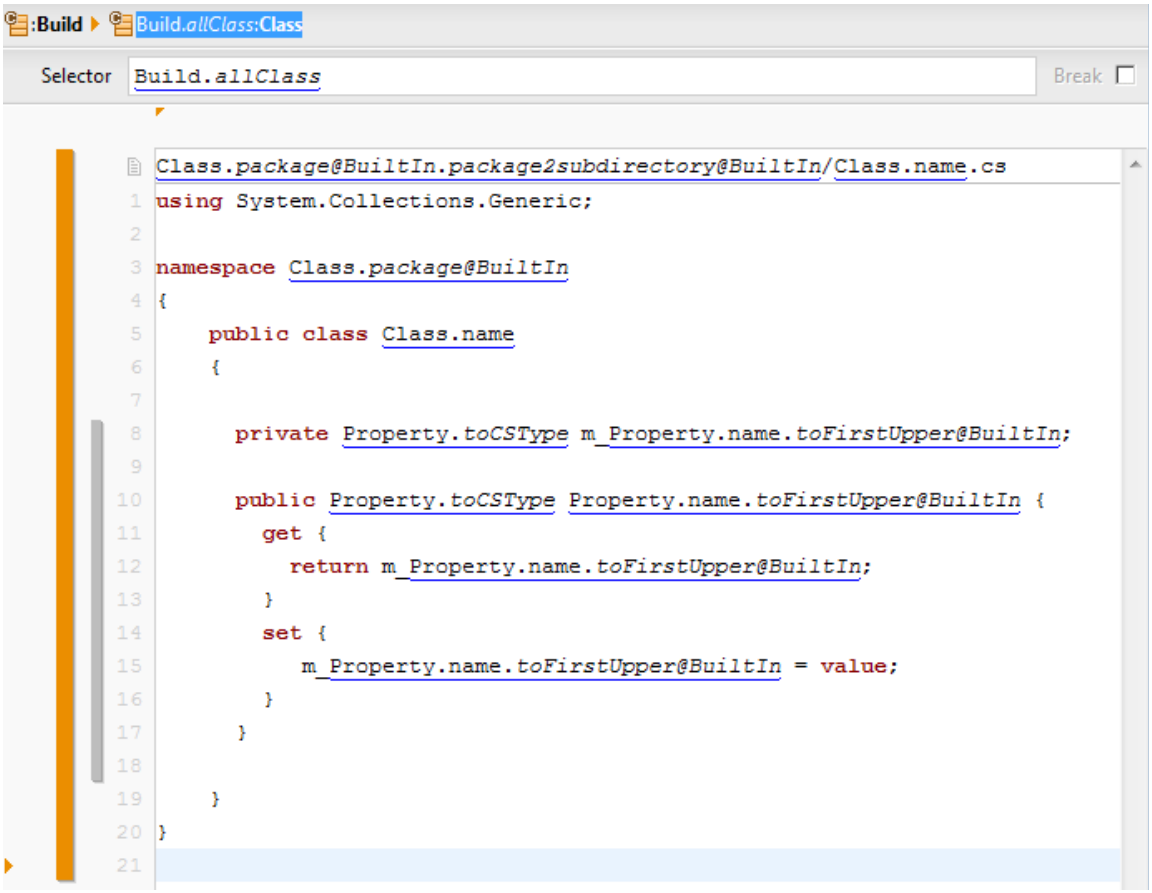


Рисунок 2. Пример генератора в системе Actifsource

При таком подходе невозможна полная генерация кода, но можно довольно успешно проводить прототипирование или создавать скелет программного обеспечения.

2.3 MetaCase MetaEdit+

MetaEdit+ — это система компании MetaCase для создания и использования предметно-ориентированных решений. Имеет интеграцию с Eclipse и Microsoft Visual Studio. Предоставляется бесплатно на 31 день.

В системе MetaEdit+ для описания генерации кода служит программа на специальном языке, в котором в противоположность вышеописанным подходам управляющие конструкции никак не выделяются, а наоборот не меняющиеся шаблонные строки обрамляются кавычками.

```
Report '!AST'
subreport '_translators' run
'Statemachine{' newline;
subreport '_events' run
subreport '_commands' run
foreach .State; orderby :Start state?; desc {
   '  State{
      Name{"' :State name; '"},' newline;
   If :Command; then
      '     Actions[' newline;
      dowhile :Command {'         "' :Name '",' newline; }
      '"
      ],' newline;
   endif;
   '     Transitions{' newline;
   dowhile ~Source>Transition {
      '         {' newline
      '            Event{"' do :Trigger {:Name} '"},' newline
      '            ToState{"' do ~Target.State {:State name;} '"}' newline
      '         },' newline;
   }
   '"}
         }
      }
   },' newline;
 }
'}'
endreport
```

Рисунок 3. Пример генератора в системе MetaEdit+

Это снижает читаемость кода, усложняет видение результата работы. Однако MetaEdit+ предоставляет разрабатывать сколь угодно сложные генераторы.

3 Реализация

Данная работа выполняется в рамках проекта QReal.

3.1 QReal

QReal[1] представляет из себя инструментальную среду визуального моделирования. Каждая диаграмма, созданная в среде, хранится в репозитории как набор элементов диаграмм с их связями и свойствами. В рамках QReal существует механизм метаредактора, который позволяет для необходимого класса задач создать подходящий визуальный язык [1]. Этот визуальный язык также задается как диаграмма, в дальнейшем называемой метамоделью языка.

Проект разрабатывается на базе кафедры системного программирования Санкт-Петербургского Университета с 2007 года.

3.2 Язык описания генераторов Geny

Язык создавался для генераторов, использующих репозиторий QReal. Объекты в нем хранятся под универсальными идентификаторами. По идентификатору через API репозитория можно получить свойства, связанные с объектом (имя, надпись, расположение на диаграмме, тип, родительские/дочерние объекты и т.д.). Некоторые из свойств являются списковыми, например, связанные с данным объекты, другие — просто текстовые поля или числа. Для списков объектов необходимо иметь конструкцию перехода по ним, а для обычных полей объектов — уметь передавать их в поток вывода. При генерации кода создается много файлов, поэтому нужна была легкая система для работы с ними.

Эти требования и легли в основу модели расширения шаблона генерации.

Основные концепции языка

- **Принцип “прямой передачи”.** Все, что находится в программе и не обрамлено управляющими конструкциями, передается в результирующий поток (например, в выходной файл). Этот принцип позволил освободить код генератора от большого количества лишних символов. Благодаря этой концепции работа над генераторами становиться максимально прозрачной по отношению к представлению итога функционирования создаваемой системы.

[1] QReal, http://qreal.ru/

- **Принцип “текущего объекта”.** В конкретный момент времени генератор кода часто обращается к атрибутам только одного объекта. Этот объект можно задавать с помощью конструкций объектных переходов, то есть конструкций позволяющих сменить объект, над которым происходит работа. “Текущий объект” есть текущий узел при обходе графа. Механизм позволяет не писать имя объекта, при возвращении его атрибутов в поток, также укорачивает объектные переходы к связанным сущностям.
- **Система заданий.** Каждый файл программы на разработанном нами языке Geny представляет из себя именованное задание. Задания могут использовать внутри себя другие. Используется “текущий объект” вызванного места. Этот механизм обеспечивает модульность программ на языке Geny, делая возможным написание более сложных генераторов. Задания организуются в проект с помощью специального файла, в котором указывается путь до сохраненной диаграммы, а потом перечисляются файлы, в которых записаны необходимые задания. Интерпретатор исполняет задание Main, которое уже вызывает другие по необходимости.
- **Управляющие строки.** Такие строки начинаются со служебной конструкции #! и необходимы для выполнения служебных операций:
    a. toFile fileName — перенаправляет генерацию блока в файл с переданным именем;
    b. foreach — служит для выполнения блока кода с объектами из некоторого списка;
    c. saveObj markName — сохраняет “текущий объект” по переданному имени;
    d. switch/case — позволяют делать условные переходы, сравнивая определенные значения с атрибутами “текущего объекта”;
    e. if %propertyName% operation %propertyValue% — выполняет сравнение по опреации (==, !=, contains) атрибута “текущего объекта” с заданным значением;
    f. {, } — отделяют блоки кода для других конструкций.
- **Управляющая конструкция внутри контекста.** Конструкция является аналогом помеченных мест для вставки в шаблонах. Она позволяет вставить представление параметра объекта или результат выполнения задания. Для передачи атрибута ”текущего объекта” в языке Geny необходимо написать название этого атрибута между @@ @@ (к примеру, если нужно имя объекта — @@name@@), если нужен параметр некоторого помеченного объекта, то нужно перед именем параметра написать имя метки и @ (@@methodA@methodVisibility@@). Вставка задания происходит командой @@!task taskName@@.
- **Система меток объектов.** Расширяет предыдущий метод, сохраняя с помощью управляющей конструкции saveObj объект по некоторому имени. В дальнейшем по имени можно возвращать свойства помеченной сущности. Это позволяет работать сразу с несколькими объектами — например, если в дальнейшем в генераторе придется пользоваться “текущим объектом” в контексте работы с другими объектами. К примеру, если

необходимо для UML-класса получить его код на C++, нужно обойти все методы класса и при генерации их реализаций нужно знать имя класса, которое является атрибутом объекта-класса. Удобно запомнить этот объект, а потом использовать с помощью метки.

Пример генератора на языке Geny

Это пример того, как описать на языке Geny генерацию прототип класса языка Java по UML-диаграмме. В примере для краткости опущена часть, отвечающая за генерацию методов класса.

```
#!Task JavaClass                                          <- Начало задания
#!/ Produce Java class from repo                          <- Комментарий
#!foreach Class in elementsByType(Class)
#!{                                                  <- Начало блока, относящегося к
                                                      элементу, выбранному по foreach
#!toFile @@name@@.java                               <- Начало записи в файл,
                                                        соотвествующий имени класса
#!{
class @@name@@ {
#!foreach FieldsContainer in children                <- Переход к принадлежащим
                                                      контейнеру элементам диаграммы
                                                      (полям класса)
#!{
#!foreach Field in children
#!{
      @@fieldVisibility@@ @@fieldType@@ @@fieldName@@; <- Управляющие конструкции,
                                                        возвращающие в выходной поток
                                                        свойства текущего объекта
#!}
#!}
}
#!}
#!}
```

Листинг 1. Пример генератора на языке Geny

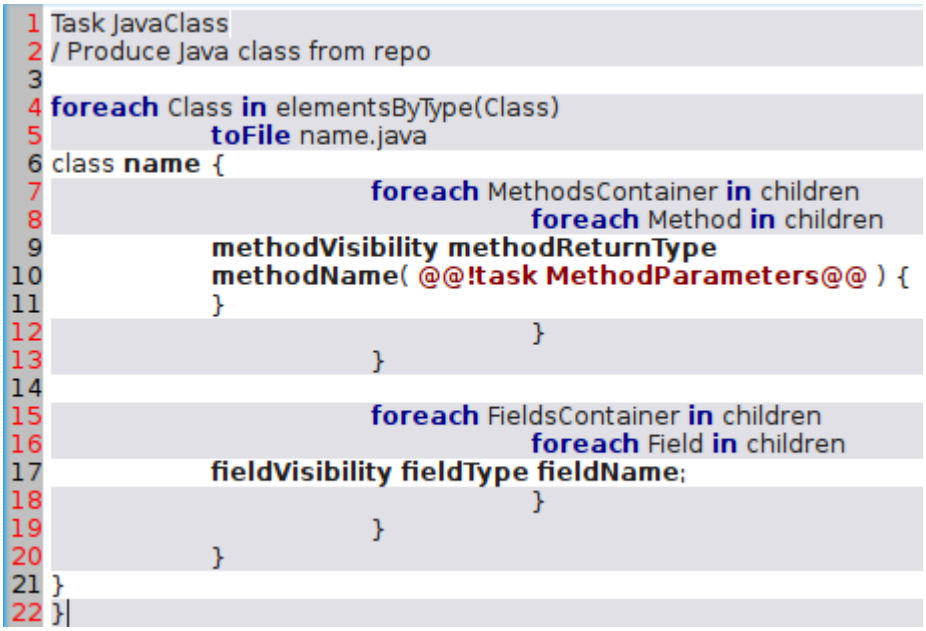
3.3 Редактор языка Geny

В процессе апробирования языка стало понятно, что необходимо разработать специализированный редактор, который позволил бы в большей степени повысить наглядность кода, упростив написание генераторов. Связано это с тем, что язык Geny содержит много малоинформативных с точки зрения пользователя префиксов (#!) и символов (@@), которые по возможности необходимо скрывать. Для более интуитивного восприятия кода генератора также крайне полезно делать цветовые акценты на управляющих конструкциях — подчеркивать их или выделять жирным шрифтом.

В итоге был разработан редактор, который внес следующие изменения в редактирование Geny-генераторов:

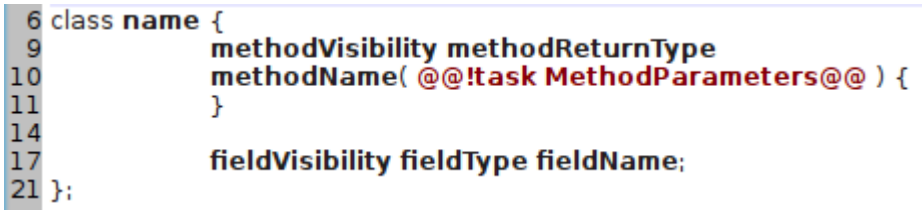
- Управляющие строки выделяются не посредством префикса #!, а с помощью цвета фона. Это позволяет разделить два языка — язык описания генератора и целевой язык генерации. На данный момент изменение типа строки происходит с помощью переключения его явно для каждой строки. В дальнейшем планируется сделать механизм, который будет анализировать введенную строку и присваивать ей тип управляющей, в случае если она похожа на конструкцию языка Geny, а пользователю необходимо будет лишь корректировать результат в случае ошибки.
- Обрамление посредством @@ управляющих конструкций, которые возвращают свойства объекта, заменено выделением жирным шрифтом, которое происходит автоматически при вводе названия свойства объекта, в случае подтверждения его автодополнением. Список доступных названий свойств для текущего генератора берется из метамодели визуального языка.
- На основе информации об управляющих блоках реализован алгоритм, который во время редактирования форматирует строки, относящиеся к коду Geny. Вложенные блоки табулируются, что также способствует повышению удобочитаемости генератора.
- Убрана необходимость писать управляющую строку #!{, которая открывает блок кода, так как она не несет полезной для пользователя информации. Для каждой управляющей строки определяется, является ли она открывающей для блока, что означает следование #!{ за ней, а значит их можно логически объединить. Так сопоставляется начало блока с закрывающей строкой #!}.
- Есть возможность скрыть управляющие строки, чтобы можно было наглядно увидеть шаблон генерации. Это позволяет иметь более четкое представление, что в итоге будет сгенерировано.

Сам редактор интегрирован в среду QReal. Проект Geny-генератора дополняет метамодель визуального языка информацией о способе трансляции диаграмм, созданными с помощью данного языка, в итоговый код, что позволяет использовать генератор при работе над такими диаграммами.



```
1  Task JavaClass
2  / Produce Java class from repo
3
4  foreach Class in elementsByType(Class)
5              toFile name.java
6  class name {
7                          foreach MethodsContainer in children
8                                      foreach Method in children
9              methodVisibility methodReturnType
10             methodName( @@!task MethodParameters@@ ) {
11             }
12                                     }
13                         }
14
15                         foreach FieldsContainer in children
16                                     foreach Field in children
17             fieldVisibility fieldType fieldName;
18                                     }
19                         }
20             }
21 }
22 }
```

Рисунок 4. Представление генератора прототипа Java-класса в редакторе



```
6  class name {
9              methodVisibility methodReturnType
10             methodName( @@!task MethodParameters@@ ) {
11             }
14
17             fieldVisibility fieldType fieldName;
21 };
```

Рисунок 5. Представление генератора прототипа Java-класса без управляющих строк

Использование редактора позволяет уменьшить количество управляющих строк примерно на треть за счет сокрытия #!{, так как почти каждая конструкция языка Geny подразумевает структуру. Замена обрамлений @@, необходимых для интеграции управляющих конструкций внутрь шаблонных строк, на выделение жирным шрифтом не только уменьшает объем кода, но и помогает ярче отделить конструкции от текста, непосредственно передаваемого в выходной поток. А режим отображения без управляющих строк дает ясное представление о том, как будет выглядеть итог работы генератора.

В данном примере количество управляющих строк сократилось с 20 до 14, было убрано 7 обрамлений @@, 14 префиксов #!. Учитывая, что размер генератора в редакторе составляет 22 строки, это является существенным улучшением.

4 Апробация

Для работы системы был создан интерпретатор, после чего развитие языка пошло путем внесения изменений в связи с решаемыми задачами. Ниже описаны примера таких задач.

4.1 Генератор заголовочных файлов C++-классов по UML-модели

В репозитории QReal UML-класс хранится вместе с двумя дочерними объектами-контейнерами для методов и полей. Так, генератор заголовочного файла должен обойти всех потомков объекта-класса, затем — объекта-контейнера и использовать их имена для генерации заголовка метода или объявления поля. Этот пример аналогичен приведенному генератору прототипов Java-классов.

На языке Geny в представлении редактора генератор занял 22 строки. Аналогичный генератор, написанный на C++, занял 170 строк без учета размера шаблонов.

4.2 Генератор языка блок-схем

Визуальный язык состоял из пяти элементов — начальная вершина, конечная вершина, действие, условный переход и связь между элементами, которая хранит для условного перехода указание на тип ветви (по положительному и отрицательному условиям). Генератор ищет начальные вершины, после чего запускается рекурсивный обход узлов диаграммы по направлению стрелок с помощью системы заданий.

Генератор на языке Geny занял около 70 строчек кода (50 в представлении редактора). Добавление в язык элемента-цикла повлекло за собой добавление в генератор еще 10 строк. Решение этой же задачи заняло более 100 строк на C++.

4.3 QReal:Robots

Был разработан генератор кода программ на C для nxtOSEK[1], являющейся платформой для роботов Lego Mindstorms NXT.

Графический язык похож на язык блок-схем. Элементы в нем разделены на четыре категории:

- **Алгоритмы.** Отвечают за поток управления программы — цикл, условный переход.
- **Действия.** Обозначают простую инструкцию роботу — подать определенную мощность на мотор, испустить сигнал.
- **Инициализация.** Проводят подготовку устройств, связывают их с портами робота.
- **Ожидания.** Приостанавливают выполнение текущей ветки потока управления до наступления определенного события (срабатывания датчика света, касания) или просто на некоторое время.

Рисунок 6. Пример диаграммы на языке описания алгоритма работы робота в системе QReal

При генерации для каждого элемента диаграммы создается отдельная процедура, в которой происходит выполнение соответствующей функциональности, а потом происходит передача управления следующим блокам.

[1] nxtOSEK, http://lejos-osek.sourceforge.net/

```
void WaitForColor0() {
	while (
		ecrobot_get_nxtcolorsensor_id(NXT_PORT_S2)
		!=
		NXT_COLOR_RED
	){
	}
	Loop12();
}
```

Листинг 2. Код по элементу “Ждать цвет”

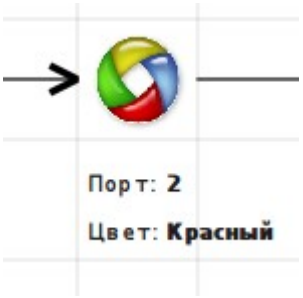


Рисунок 7.1. Элемент “Ждать цвет”

```
1  Task WaitForColorElement
2  		while (
3  			ecrobot_get_nxtcolorsensor_id(NXT_PORT_SPort)
4  			!=
5  if %Color% == %Красный%
6  			NXT_COLOR_RED
7  }
8  if %Color% == %Зелёный%
9  			NXT_COLOR_GREEN
10 }
11 if %Color% == %Синий%
12 			NXT_COLOR_BLUE
13 }
14 if %Color% == %Чёрный%
15 			NXT_COLOR_BLACK
16 }
17 if %Color% == %Жёлтый%
18 			NXT_COLOR_YELLOW
19 }
20 if %Color% == %Белый%
21 			NXT_COLOR_WHITE
22 }
23 		)
24 		{
25 		}
26 @@!task NextElementCall@@
```

Рисунок 7.2. Задание на Geny для элемента “Ждать цвет”

Код листинга 2 генерируется для элемента, который означает ожидание срабатывания датчика цвета на красный (рис. 7.1). Датчик подключен ко второму порту на роботе. Соответствующее задание из Geny-генератора приведено на рисунке 7.2.

```
void Loop12() {
	for (int __iter__ = 0; __iter__ < 10; __iter__++) {
		Timer13();
	}
	PlayTone4();
}
```

Листинг 3. Код по элементу “Цикл”

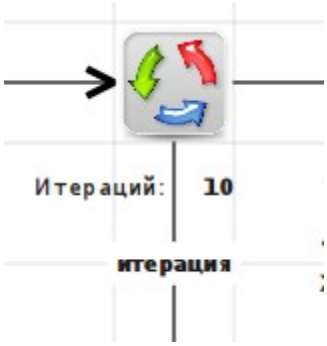


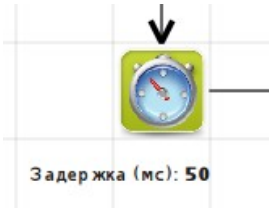
Рисунок 8.1. Элемент “Цикл”

```
1  Task LoopElement
2  saveObj LOOP
3  foreach . in outgoingLinks
4                  if %Guard% == %итерация%
5                                  for to
6                  for (int __iter__ = 0; __iter__ < Iterations; __iter__++) {
7                                  uniqueName();
8                  }
9                                  }
10                 }
11 }
12 / For nextElement
13 foreach . in outgoingLinks
14                 if %Guard% == %%
15                                 for to
16                 uniqueName();
17                                 }
18                 }
19 }
```

Рисунок 8.2. Задание на Geny для элемента “Цикл”

Блоку цикла (рис. 8.1) в коде соответствует метод из листинга 3. Происходит десятикратный вызов поддиаграммы, соответствующей ветке “итерация”, а потом осуществляется переход по другой ветке. Geny-задание на рисунке 8.2.

```
void Timer13() {
	systick_wait_ms(50);
	Beep9();
}
```



Листинг 4. Код по элементу “Таймер” Рисунок 9. Элемент “Таймер”

Листинг 4 представляет код, получаемый по таймеру с рисунка 9. В момент срабатывания таймера происходит задежка выполнения текущей ветки алгоритма на 50 миллисекунд.

На данный момент остается открытой проблема с генерацией кода, соответствующему блоку “Функция”. Этот элемент диаграммы позволяет вставлять С-подобные инструкции. В его синтаксисе для снятия значения с сенсоров дистанции используется конструкция “Сенсор1”, “Сенсор2” и т.д. В коде, который получается по диаграмме, эти выражения должны быть заменены на `ecrobot_get_sonar_sensor(NXT_PORT_S1)` и `ecrobot_get_sonar_sensor(NXT_PORT_S2)` соответственно. Пока в Geny нет конструкций, позволяющих изменять подстроки возвращаемых свойств объекта, но планируется в ближайшем будущем добавить словарь замен, который даст возможность справиться с этой проблемой. Также это позволит в некоторых случаях сократить код генератора. Так задание с рисунка 7.2 могло бы выглядеть как на рисунке 10.

```
1 Task WaitForColorElement
2 		while (ecrobot_get_nxtcolorsensor_id(NXT_PORT_SPort)
3 				!= Color)
4 		{
5 		}
6 @@!task NextElementCall@@
```

Рисунок 10. Задание на Geny со словарем замен для элемента “Ждать цвет”

Объем в строчках кода генератора на C++ - 1440, на языке Geny - 310.

Заключение

В рамках данной работы был проведен обзор реализованных подходов к решению поставленной задачи, в результате которого были выявлены пути для улучшения существующих моделей разработки генераторов. На основе сделанных выводов были разработаны язык описания генераторов Geny и интерпретатор к нему, которые апробировались на примерах и корректировались по результатам тестирования. После чего было принято решение реализовать специализированный редактор для созданного языка с целью упростить использование конструкций Geny. Данный редактор позволил сократить объем кода генераторов, улучшить их восприятие.

В дальнейшем планируется переписать часть внутренних генераторов QReal с C++ на Geny, модернизировать язык, в соответствии с требованиями, которые будут возникать при использовании языка. Также в будущем необходимо будет провести более тесную интеграцию редактора с другими DSM-инструментами QReal, в первую очередь с метаредактором. В частности, добавить возможность автоматического обновления генератора при изменении метамодели для некоторых простых случаев, таких как переименование объектов, свойств.

Список литературы

[1] Кузенкова А.С., Дерипаска А.О., Таран К.С., Подкопаев А.В., Литвинов Ю.В., Брыксин Т.А., Средства быстрой разработки предметно-ориентированных решений в metaCASE-средстве QReal // Научно-технические ведомости СПбГПУ, Информатика, телекоммуникации, управление. Вып. 4 (128). СПб.: Изд-во Политехнического Университета. 2011, С. 142-145.

[2] M. Mazaud, R. Rakotozafy, and A. Szumachowski-Despland. 1987. Code generator generation based on template-driven target term rewriting. In on Rewriting techniques and applications, Pierre Lescanne (Ed.). Springer-Verlag, London, UK, 105-120.

[3] Raphael Mannadiar and Hans Vangheluwe. 2010. Domain-specific engineering of domain-specific languages. In Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM '10). ACM, New York, NY, USA, , Article 11 , 6 pages

[4] M.Riser, R.Carrara Modelle im Rampenlicht: Voraussetzungungen für agiles Arbeiten mit Modellen, OO-Magazin OBJEKTspektrum, June 2010.

[5] Tolvanen, J.-P., Making model-based code generation work - Practical examples (Part 2), Embedded Systems Europe, Vol. 9, 64 (March), 2005